

10/544894

REC'D PTO 05 AUG 2003

REC'D 03 MAY 2004

WIPO

PCT

P1 1159474

# THE UNITED STATES OF AMERICA

**TO ALL TO WHOM THESE PRESENTS SHALL COME:**

**UNITED STATES DEPARTMENT OF COMMERCE  
United States Patent and Trademark Office**

**April 28, 2004**

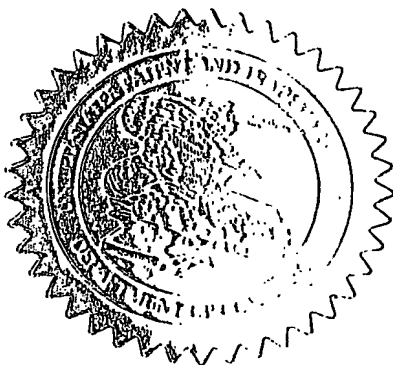
**THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM  
THE RECORDS OF THE UNITED STATES PATENT AND TRADEMARK  
OFFICE OF THOSE PAPERS OF THE BELOW IDENTIFIED PATENT  
APPLICATION THAT MET THE REQUIREMENTS TO BE GRANTED A  
FILING DATE.**

**APPLICATION NUMBER: 60/445,339**

**FILING DATE: February 05, 2003**

**RELATED PCT APPLICATION NUMBER: PCT/US04/03609**

**By Authority of the  
COMMISSIONER OF PATENTS AND TRADEMARKS**



*T. Wallace*  
**T. WALLACE**  
Certifying Officer

**PRIORITY  
DOCUMENT**

**SUBMITTED OR TRANSMITTED IN  
COMPLIANCE WITH RULE 17.1(a) OR (b)**

02-07-03 604-45339-02048032

02/05/03  
JC914 U.S. PTO

PTO/SB/16 (10/01)  
Approved for use through 10/31/2002. OMB 0551-0032  
U.S. Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE  
Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

# PROVISIONAL APPLICATION FOR PATENT COVER SHEET

This is a request for filing a PROVISIONAL APPLICATION FOR PATENT under 37 CFR 1.53(c).

Express Mail Label No. EL919125687US

JC971 U.S. PTO  
60/445339  
02/05/03

INVENTOR(S)					
Given Name (first and middle (if any))	Family Name or Surname	Residence (City and either State or Foreign Country)			
Aravind R. Ali Sethuraman	Dasu Akoglu Panchanathan	Tempe, Arizona Tempe, Arizona Gilbert, Arizona			
<input type="checkbox"/> Additional inventors are being named on the _____ separately numbered sheets attached hereto					
TITLE OF THE INVENTION (500 characters max)					
Reconfigurable Processing					
Direct all correspondence to: CORRESPONDENCE ADDRESS					
<input checked="" type="checkbox"/> Customer Number		28,529		Place Customer Number Bar Code Label here	
OR					
<input type="checkbox"/> Firm or Individual Name					
Address					
Address					
City		State	ZIP		
Country		Telephone	Fax		
ENCLOSED APPLICATION PARTS (check all that apply)					
<input checked="" type="checkbox"/> Specification Number of Pages		198		<input type="checkbox"/> CD(s), Number	
<input type="checkbox"/> Drawing(s) Number of Sheets				<input checked="" type="checkbox"/> Other (specify)	
<input type="checkbox"/> Application Data Sheet. See 37 CFR 1.76					
METHOD OF PAYMENT OF FILING FEES FOR THIS PROVISIONAL APPLICATION FOR PATENT					
<input checked="" type="checkbox"/> Applicant claims small entity status. See 37 CFR 1.27.				FILING FEE AMOUNT (\$)	
<input checked="" type="checkbox"/> A check or money order is enclosed to cover the filing fees				80	
<input checked="" type="checkbox"/> The Commissioner is hereby authorized to charge filing fees or credit any overpayment to Deposit Account Number:				070135	
<input type="checkbox"/> Payment by credit card. Form PTO-2038 is attached.					
The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.					
<input checked="" type="checkbox"/> No.					
<input type="checkbox"/> Yes, the name of the U.S. Government agency and the Government contract number are: _____					

Respectfully submitted,

SIGNATURE

TYPED or PRINTED NAME

TELEPHONE

Thomas D. MacBlain

Date

REGISTRATION NO.  
(if appropriate)  
Docket Number.

24,583

9138-0106

## USE ONLY FOR FILING A PROVISIONAL APPLICATION FOR PATENT

This collection of information is required by 37 CFR 1.51. The information is used by the public to file (and by the PTO to process) a provisional application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.14. This collection is estimated to take 8 hours to complete, including gathering, preparing, and submitting the complete provisional application to the PTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, Washington, D.C. 20231. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Box Provisional Application, Assistant Commissioner for Patents, Washington, D.C. 20231.

6044 339 020503

PTO/SB/17 (01-03)

Approved for use through 04/30/2003. OMB 0651-0032  
U.S. Patent and Trademark Office; U.S. DEPARTMENT OF COMMERCE

Under the Paperwork Reduction Act of 1995, no persons are required to respond to a collection of information unless it displays a valid OMB control number.

**FEE TRANSMITTAL  
for FY 2003**

Effective 01/01/2003, Patent fees are subject to annual revision.

☒ Applicant claims small entity status. See 37 CFR 1.27

TOTAL AMOUNT OF PAYMENT (\$ ) 80

**Complete if Known**

Application Number	
Filing Date	herewith
First Named Inventor	Dasu
Examiner Name	
Art Unit	
Attorney Docket No.	9138-0106

**METHOD OF PAYMENT (check all that apply)**☒ Check ☐ Credit card ☐ Money Order ☐ Other ☐ None☒ Deposit Account:Deposit  
Account  
Number  
Deposit  
Account  
Name

070135

Gallagher &amp; Kennedy, P.A.

The Commissioner is authorized to: (check all that apply)

☐ Charge fee(s) indicated below ☒ Credit any overpayments  
☒ Charge any additional fee(s) during the pendency of this application  
☐ Charge fee(s) indicated below, except for the filing fee to the above-identified deposit account.
**FEE CALCULATION****1. BASIC FILING FEE**

Large Entity Fee Code (\$)	Small Entity Fee Code (\$)	Fee Description	Fee Paid
1001 750	2001 375	Utility filing fee	
1002 330	2002 165	Design filing fee	
1003 520	2003 260	Plant filing fee	
1004 750	2004 375	Reissue filing fee	
1005 160	2005 80	Provisional filing fee	80

SUBTOTAL (1) (\$ ) 80

**2. EXTRA CLAIM FEES FOR UTILITY AND REISSUE**

Total Claims	Extra Claims	Fee from below	Fee Paid
Independent Claims	-20** =	X	
Multiple Dependent	-3** =	X	

Large Entity Fee Code (\$)	Small Entity Fee Code (\$)	Fee Description	Fee Paid
1202 18	2202 9	Claims in excess of 20	
1201 84	2201 42	Independent claims in excess of 3	
1203 280	2203 140	Multiple dependent claim, if not paid	
1204 84	2204 42	** Reissue independent claims over original patent	
1205 18	2205 9	** Reissue claims in excess of 20 and over original patent	

SUBTOTAL (2) (\$ )

\*\*or number previously paid, if greater; For Reissues, see above

**FEE CALCULATION (continued)****3. ADDITIONAL FEES**

Large Entity Small Entity

Fee Code (\$)	Fee Code (\$)	Fee Description	Fee Paid
1051 130	2051 65	Surcharge - late filing fee or oath	
1052 50	2052 25	Surcharge - late provisional filing fee or cover sheet	
1053 130	1053 130	Non-English specification	
1812 2,520	1812 2,520	For filing a request for <i>ex parte</i> reexamination	
1804 920*	1804 920*	Requesting publication of SIR prior to Examiner action	
1805 1,840*	1805 1,840*	Requesting publication of SIR after Examiner action	
1251 110	2251 55	Extension for reply within first month	
1252 410	2252 205	Extension for reply within second month	
1253 930	2253 465	Extension for reply within third month	
1254 1,450	2254 725	Extension for reply within fourth month	
1255 1,970	2255 985	Extension for reply within fifth month	
1401 320	2401 160	Notice of Appeal	
1402 320	2402 160	Filing a brief in support of an appeal	
1403 280	2403 140	Request for oral hearing	
1451 1,510	1451 1,510	Petition to institute a public use proceeding	
1452 110	2452 55	Petition to revive - unavoidable	
1453 1,300	2453 650	Petition to revive - unintentional	
1501 1,300	2501 650	Utility issue fee (or reissue)	
1502 470	2502 235	Design issue fee	
1503 630	2503 315	Plant issue fee	
1460 130	1460 130	Petitions to the Commissioner	
1807 50	1807 50	Processing fee under 37 CFR 1.17(q)	
1808 180	1808 180	Submission of Information Disclosure Stmt	
8021 40	8021 40	Recording each patent assignment per property (times number of properties)	
1809 750	2809 375	Filing a submission after final rejection (37 CFR 1.129(a))	
1810 750	2810 375	For each additional invention to be examined (37 CFR 1.129(b))	
1801 750	2801 375	Request for Continued Examination (RCE)	
1802 900	1802 900	Request for expedited examination of a design application	

Other fee (specify)

\*Reduced by Basic Filing Fee Paid

SUBTOTAL (3) (\$ )

**SUBMITTED BY**

Name (Print/Type) Thomas D. MacBlain

Registration No. 24,583

(Attorney/Agent)

Telephone 602 530-8088

Signature

Date

2/5/03

**WARNING: Information on this form may become public. Credit card information should not be included on this form. Provide credit card information and authorization on PTO-2038.**

This collection of information is required by 37 CFR 1.17 and 1.27. The information is required to obtain or retain a benefit by the public which is to file (and by the USPTO to process) an application. Confidentiality is governed by 35 U.S.C. 122 and 37 CFR 1.14. This collection is estimated to take 12 minutes to complete, including gathering, preparing, and submitting the completed application form to the USPTO. Time will vary depending upon the individual case. Any comments on the amount of time you require to complete this form and/or suggestions for reducing this burden, should be sent to the Chief Information Officer, U.S. Patent and Trademark Office, U.S. Department of Commerce, Washington, DC 20231. DO NOT SEND FEES OR COMPLETED FORMS TO THIS ADDRESS. SEND TO: Commissioner for Patents, Washington, DC 20231.

If you need assistance in completing the form, call 1-800-PTO-9199 (1-800-786-9199) and select option 2.

## IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant: Dasu et al.

Filed: Herewith

For Provisional Application Titled: Reconfigurable Processing

---

**CERTIFICATE OF MAILING BY EXPRESS MAIL**  
**"Express Mail" mailing label number EL919125687US**

---

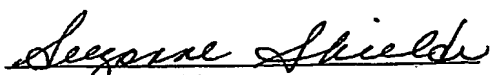
Commissioner for Patents  
Box Provisional Patent Application  
Washington, D.C. 20231

Dear Commissioner:

I hereby certify that the following correspondence is being deposited in the United States Postal Service as Express Mail on the date shown below in an envelope addressed as shown above.

1. Provisional Application for Patent Cover Sheet (1 sheet);
2. Fee Transmittal for FY 2003 (1 sheet in duplicate);
3. Specification 198 pages including:
  - cover page, and claims (3 pages),
  - "Reconfigurable Processing" (41 pages);
  - Article "Pattern Recognition Tool to Detect..." (8 pages),
  - "Reconfigurable Tool Set Development" (48 pages),
  - "Relevant Work" (6 pages),
  - Article "Efficient Path Profiling" (12 pages),
  - Article "Speeding up Program Execution Using ..." (5 pages),
  - Article "Configuration Code Generation and ..." (12 pages),
  - Article "Reconfigurable Processing: The Solution to Low-Power ..." (4 pages),
  - Article "Instruction Generation and Regularity Extraction for ..." (8 pages),
  - Article "Automatic Detection of Recurring Operation Patterns" (5 pages),
  - Article "Matching and Covering with Multiple-Output Patterns" (30 pages),
  - Article "Instruction Set Synthesis Using Operation Pattern Detection (8 pages),
  - Publication Chameleon Systems, Inc. "Wireless Base Station Design..." (8 pages);
4. Check for \$80.00; and
5. A return receipt postcard.

Date

2/5/03  
Suzanne ShieldsGALLAGHER & KENNEDY, P.A.  
2575 East Camelback Road  
Phoenix, Arizona 85016-9255  
Tel. No. (602) 530-8000  
Fax. No. (602) 530-8500  
1044481v4

6044539.020503

Attorney Docket No. 9138-0106

Express Mail Label No. EL919125687US

**IN THE UNITED STATES PATENT AND TRADEMARK OFFICE**

**Provisional Patent Application**

**Title:** Reconfigurable Processing

**Inventors:** Aravind R. Dasu  
Ali Akoglu  
Sethuraman Panchanathan

# Reconfigurable Processing

Aravind Dasu

## Abstract

Multimedia processing is becoming increasingly important with wide variety of applications ranging from multimedia cell phones to high definition interactive television. Media processing involves the capture, storage, manipulation and transmission of multimedia objects such as text, handwritten data, audio objects, still images, 2D/3D graphics, animation and full-motion video. A number of implementation strategies have been proposed for processing multimedia data. These approaches can be broadly classified based on the evolution of processing architectures and the functionality of the processors. In order to provide media processing solutions to different consumer markets, designers have combined some of the classical features from both the functional and evolution based classifications resulting in many hybrid solutions. We have performed a detailed complexity analysis of the recent multimedia standard (MPEG-4) which has shown the potential for reconfigurable computing, that adapts the underlying hardware dynamically in response to changes in the input data or processing environment. We therefore propose a methodology for designing a reconfigurable media processor. This involves the design of a parser, profiler, recurring pattern analyzer, spatial and temporal partitioner. The proposed methodology enables efficient partitioning of resources for complex and time critical multimedia applications.

## 1 Introduction

A variety of media processing techniques are typically used in multimedia processing environments to capture, store, manipulate and transmit multimedia objects such as text, handwritten data, audio objects, still images, 2D/3D graphics, animation and full-motion video. Example techniques include speech analysis and synthesis, character recognition, audio compression, graphics animation, 3D rendering, image enhancement and restoration, image/video analysis and editing, and video transmission. Multimedia computing presents challenges from the perspectives of both hardware and software. For example, multimedia standards such as MPEG-1, MPEG-2, MPEG-4, MPEG-7, H.263 and JPEG 2000 involve execution of complex media processing tasks in real-time. The need for real-time processing of complex algorithms is further accentuated by the increasing interest in 3-D image and stereoscopic video processing. Each media in a multimedia environment requires different processes, techniques, algorithms and hardware. The complexity, variety of techniques and tools, and the high computation, storage and I/O bandwidths associated with multimedia processing presents opportunities for reconfigurable processing to enable features such as scalability, maximal resource utilization and real-time implementation.

A number of implementation strategies have been proposed for processing multimedia data. These approaches can be broadly classified based on evolution of media processing architectures and functionality. In order to provide media processing solutions to different consumer markets, designers have combined some of the classical features from both the functional and evolution based classifications resulting in many hybrid solutions. In this

paper, we have proposed a categorization through a judicious mixture of features from both classifications. It is shown that the complexity, real time constraints and the need for low power, area and cost efficient implementations cannot all be satisfied by the existing solution strategies, therefore the need to explore the paradigm of reconfigurable computing. We propose a methodology for designing a reconfigurable media processor. This involves hardware-software co-design implemented in the form of a parser, profiler, recurring pattern analyzer, spatial and temporal partitioner. The proposed methodology enables efficient partitioning of resources for complex and time critical multimedia applications. To demonstrate the potential for reconfiguration in multimedia computations, we have performed a detailed complexity analysis of the recent multimedia standard (MPEG-4), which we believe involves multiple media and encompasses a wide range of operations typically found in media processing. The results of our analysis show that there are significant variations in the computational complexity among the various modes/operations of MPEG-4. This points to the potential for extensive opportunities for exploiting reconfigurable implementations of multimedia algorithms.

This document is organized as follows. Section 2 gives an overview of the classification of existing media processing approaches. Section 3 provides details on general-purpose programmable processors. Section 4 discusses the state of the art media processor architectures and Section 5 discusses the dedicated hardware implementations. Section 6 justifies the need for a reconfigurable system design and the proposed methodology to design a dynamically reconfigurable multimedia processor.

## **2 Media Processing Approaches**

The factors that influence the design of cost effective media processing solutions are as follows:

- Detailed analysis of the computational complexity, variety of techniques and tools associated with multimedia processing.
- Evaluation of real time constraints of typical media processing applications, including task switching between the various media components.
- Investigation of the parallelism and redundancies associated with media algorithms, stemming from their repetitive and compute intensive nature.
- Understanding of the inter-processor communication patterns for System on Chip (SOC) and Chipset based implementations.
- Exploration of memory issues ranging from fast high density and expensive on-chip RAMs to low cost high speed off chip RAMs.
- Evaluation of the cost/time tradeoff involved in providing augmentation to existing solutions in order to support media processing.
- Investigation of the trade offs between processing power and processor-memory bandwidth for a restricted area and low power implementation.

Existing media processing strategies and solutions can be classified based on either the evolution of processing architectures or the functionality of the processors. Section 2.1 discusses the classification based on the former approach while section 2.2 categorizes the processors based on the latter approach.

## 2.1 A Classification:

In order to provide media processing solutions to different consumer markets, designers have combined some of the classical features from both the functional and evolution based classifications resulting in many hybrid solutions. We recall from section 2 that a number of factors affect the design of media processing architectures. We propose a hybrid or a combinational approach to classifying existing media processing solutions, based on the dominating processing flavours.

From an evolutionary standpoint, special purpose programmable processors assimilate the features of DSP and RISC based architectures. But from a functional perspective, they incorporate units that exploit parallelism at many levels, thus adding the flavors of VLIW, SIMD onto the processing core. Therefore based on the dominating factors, we propose a hybrid classification of the multimedia processing solutions (Figure 1).

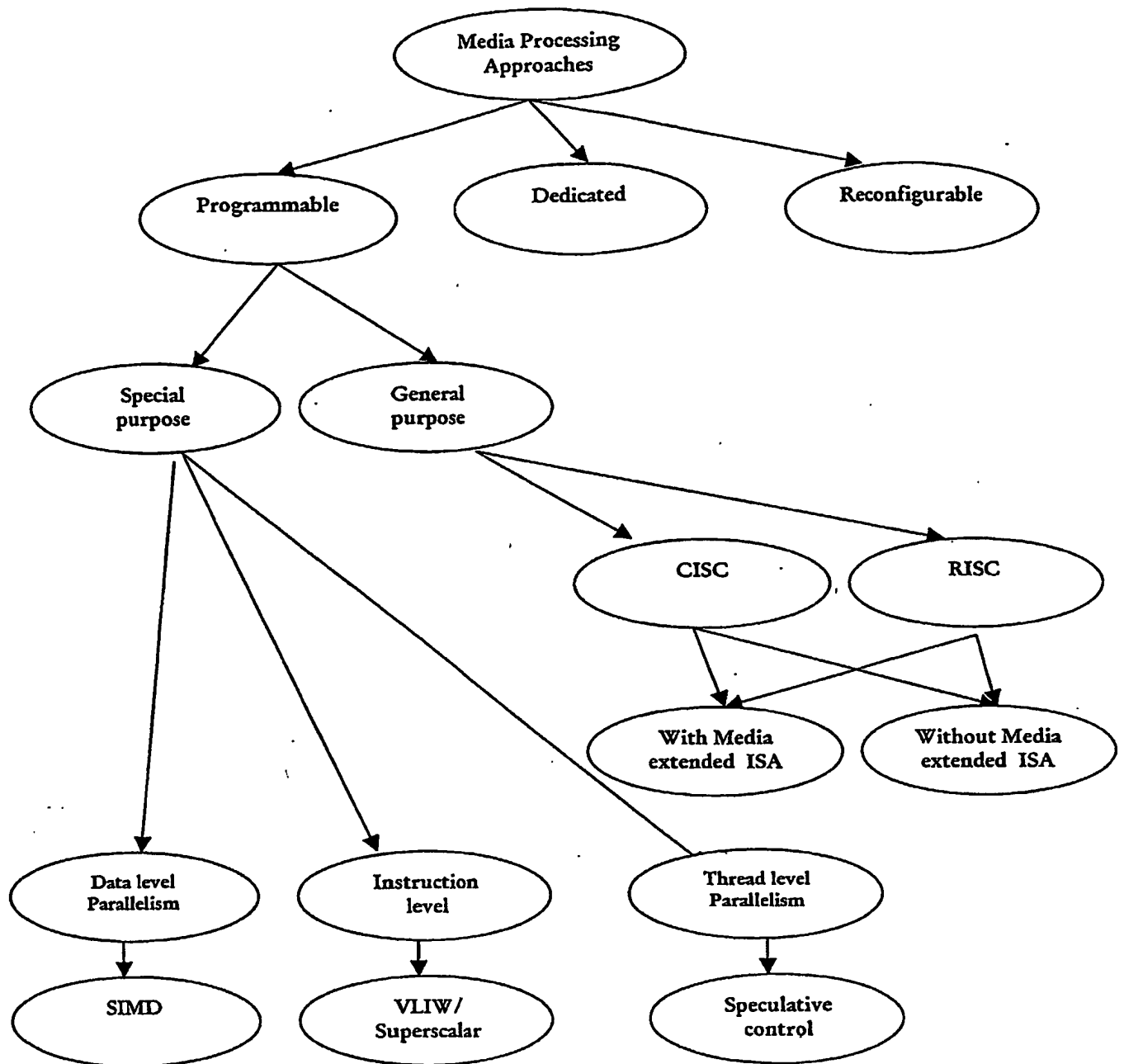


Figure 1 A hybrid classification

The details of processors that are categorized based on the above classification are presented in-section 3.

### 3 General Purpose programmable Processors:

CISC and RISC processors originally intended for general purpose computing have also found a niche in processing multimedia data. There also exist a class of microprocessors, which are used for media processing applications, but without explicit support through their instruction set architectures. Examples of such processors are Crusoe family [2] (by Transmeta corporation) and Intel's Strong-ARM processor family [3]. These processors compensate for the lack of specialized support by enhancing the computing power using technologies like code morphing, VLIW based instruction issue and clock gating. The need for backward compatibility with the large number of existing processors such as Pentium and microSPARC-II led to the design of media specific extensions to their Instruction Set Architectures (ISAs). Examples of CISC processors with media support include Intel's Pentium 4 with SSE [4] and AMD's Duron with 3D Now! [5], [6], [7]. RISC processors with such support include UltraSPARC-I with VIS [8] and Power PC with AltiVec [9].

Details of the CISC and RISC based architectures as follows:

#### 3.1 RISC architecture:

The instruction set of the RISC class of processors is characterized by the most frequently used instructions for general purpose computing. More complex instructions or less frequently used instructions are implemented as sequences of the reduced instruction set. The new generation of processors such as the Sun UltraSPARC (termed as *super scalar*) [31], [32], [33] issue upto 4 instructions simultaneously per cycle. This feature exploits Instruction Level Parallelism. To cater to the specific needs for processing multimedia data, the SPARC family of microprocessors implement the SPARC ISA version 9, a 64-bit ISA with a multimedia extension called VIS [34]. These instructions are used for specialized pixel operations that can operate in parallel on 8-bit, 16-bit or 32-bit integer values packed in a 64-bit floating point register. It also includes instructions for certain 3D to 2D conversions, edge processing, data alignment, pixel distance, packing, etc. Only three percent of the actual chip real estate is assigned to the graphics instructions. These instructions are executed by the Floating point/Graphics units (Figure 2).

The FGU is composed of the following five functional units:

- floating-point divide/square root
- floating-point addition/subtraction/absolute value/negative
- floating-point multiplication
- graphics addition, align, merge, expand, and logical
- graphics multiply, pack, compare and pdist (pixel distance)

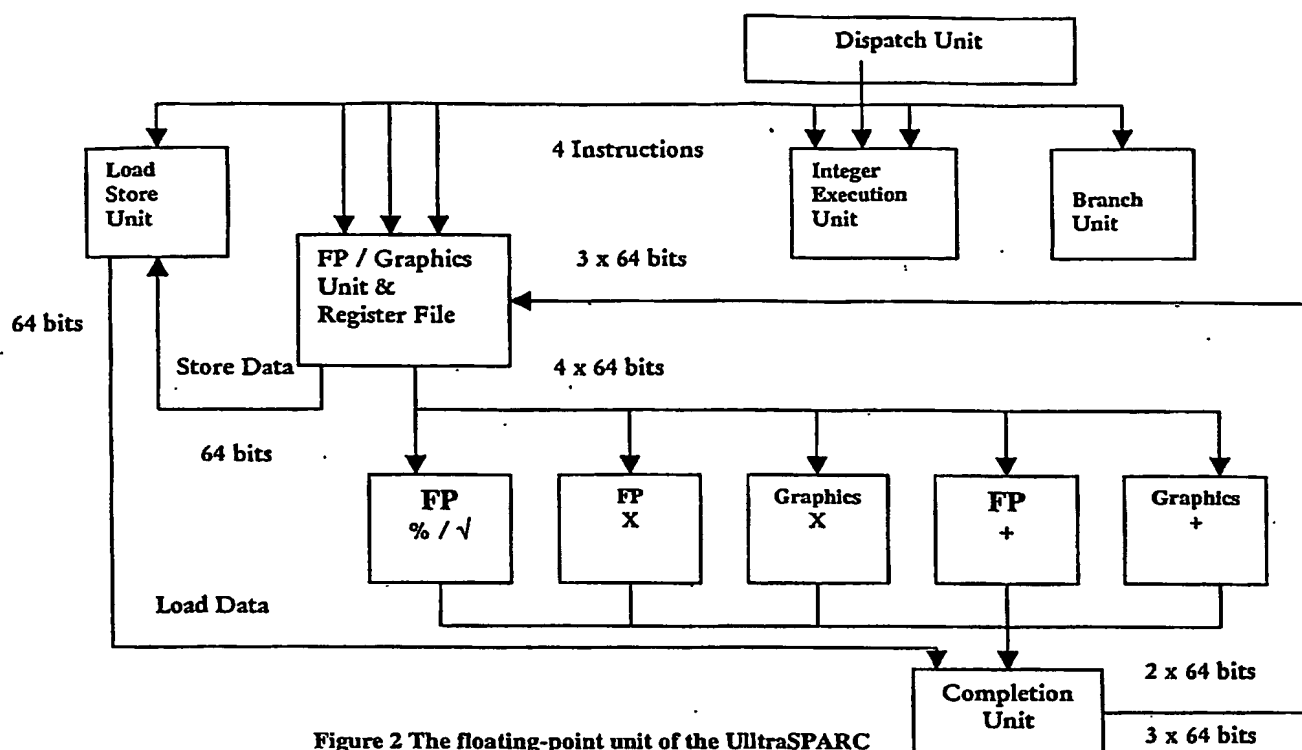


Figure 2 The floating-point unit of the UltraSPARC

Several other RISC based designs have exploited the concept of sub-word parallelism to enhance media processing [35], [36], [37]. Other leading vendors and designers of RISC processors with support for media processing include ARM processors, PowerPC's AltiVec [38] and the PA-RISC 2.0 architectures [39].

Motorola's AltiVec technology enhances the performance of the PowerPC architecture through the addition of a 128-bit vector execution unit, which operates concurrently with existing integer and floating point units. This unit allows simultaneous execution of up to 16 operations in a single clock cycle. It includes a separate register file containing 32-entries, each 128-bits wide. These 128-bit wide registers hold the data sources for the execution units. Registers are loaded and unloaded through vector store and vector load instructions that transfer the contents of a single 128-bit register to and from memory.

Vectors can be 4, 8 or 16 elements long. It uses the SIMD feature for exploiting data level parallelism. AltiVec technology offers support for:

- 16-way parallelism for 8-bit signed and unsigned integers and characters
- 8-way parallelism for 16-bit signed and unsigned integers
- 4-way parallelism for 32-bit signed and unsigned integers and IEEE floating-point numbers

Each AltiVec instruction specifies up to three source operands and a single destination operand. All operands are vector registers, with the exception of the load and store instructions and a few instruction types that provide operands from immediate fields within the instruction. 162 new unique instructions are defined for the AltiVec technology. These instructions can be classified into the following major classes:

- Intra-element arithmetic operations
- Intra-element non-arithmetic operations
- Inter-element arithmetic operations
- Inter-element non-arithmetic operations

A combination of these features enables the AltiVec enabled PowerPC processors to accelerate multimedia applications.

The ARM7500FE [40], a 32-bit RISC Network computing multimedia processor based on a cached ARM7 32-bit core has memory and I/O controllers, three DMA channels and stereo audio ports. The on-chip video controller, which includes a color palette, can directly drive either a CRT or LCD display. Migrating from still image processing to motion based video processing, several processors [41], [42], have been designed with instruction sets specifically tailored to accelerate MPEG based video sequences.

### 3.2 CISC Architecture:

In contrast to the reduced instruction set architectures, the CISC trend has always been towards more complicated and feature rich ISAs. This has been mainly due to the introduction of high-level languages and the subsequent effort towards minimizing the semantic gap between the HLL constructs and the machine instruction set. Intel, AMD and VIA are examples of the manufacturers of CISC processors. Intel's Pentium family enhances media rich application execution through support for media specific operations by using MMX and SSE [43], [44] technologies.

Since floating-point computation is at the heart of multimedia rich operations such as 3D geometry, speeding up floating-point computation is vital to enhancing overall 3D performance. To provide an enhanced performance in graphics applications, Intel's 32-bit processors-based on the IA-32 architecture-required an increase of 1.5 to 2 times the native floating-point performance. It is observed that 3D applications can execute faster by differentiating between data used repeatedly and streaming data (data used only once and then discarded). The Pentium III's new floating-point extension lets programmers designate data as streaming and provides instructions that handle this data efficiently.

Intel's new Pentium 4 processor [45] based on the Intel-NetBurst micro-architecture has some key features that allow the Pentium 4 processor to have superior floating-point and multi-media performance. The *Floating-Point (FP) execution cluster* of the Pentium 4 processor executes the floating-point, MMX, SSE, and SSE2 instructions.

- These instructions have operands ranging from 64 to 128 bits in width.
- Many FP/multi-media applications have a fairly balanced set of multiplies and adds. The FP adder can execute one Extended-Precision (EP) addition, one Double-Precision (DP) addition, or two Single-Precision (SP) additions every clock cycle. This gives a peak 6 GFLOPS for single precision or 3 GFLOPS for double precision floating-point at 1.5 GHz.
- Many multi-media applications interleave adds, multiplies, and pack/unpack/shuffle operations. For integer SIMD operations, which are the 64-bit wide MMX or 128-bit wide SSE2 instructions, there are three execution units that run parallel. The SIMD integer ALU execution hardware can process 64 SIMD integer bits per clock cycle.
- A separate shuffle/unpack execution unit can also process 64 SIMD integer bits per clock cycle. MMX/SSE2 SIMD integer multiply instructions use the FP multiply

hardware mentioned above to also do a 128-bit packed integer multiply  $\mu$ op every two clock cycles.

- The FP divider executes all divide, square root, and remainder micro-operations ( $\mu$ ops). It is based on a double-pumped SRT radix-2 algorithm, producing two bits of quotient (or square root) every clock cycle.

Achieving significantly higher floating-point and multi-media performance requires much more than just fast execution units. It requires a balanced set of capabilities that work together. These programs often have many long latency operations in their inner loops.

- The deep buffering of the Pentium 4 processor (126  $\mu$ ops and 48 loads in flight) allows the machine to examine large sections of the program to determine the dependencies.
- The out-of-order-execution hardware often unrolls the inner execution loop of these programs numerous times in its execution window. This dynamic unrolling allows the Pentium 4 processor to overlap the long-latency FP/SSE and memory instructions.

Another CISC processor vendor, AMD has recently introduced the 3D Now! Technology [46] that enhances media performance of the Athlon and Duron family of processors.

The AMD-K6-2 microprocessor is the first implementation of AMD 3DNow!. It is a set of 21 new instructions designed to open the traditional processing bottlenecks for floating-point-intensive and multimedia applications. The following is a brief description of the technology:

The Graphics Pipeline consists of the following four stages.

- **Physics:** The CPU performs floating-point-intensive physics calculations to create simulations of the real world and the objects in it.
- **Geometry:** Next, the CPU transforms mathematical representations of objects into three-dimensional representations, using floating point intensive 3D geometry.
- **Setup:** The CPU starts the process of creating the perspective required for a 3D view, and the graphics accelerator completes it.
- **Rendering:** Finally, the graphics accelerator applies realistic textures to computer-generated objects, using per-pixel calculations of color, shadow, and position.

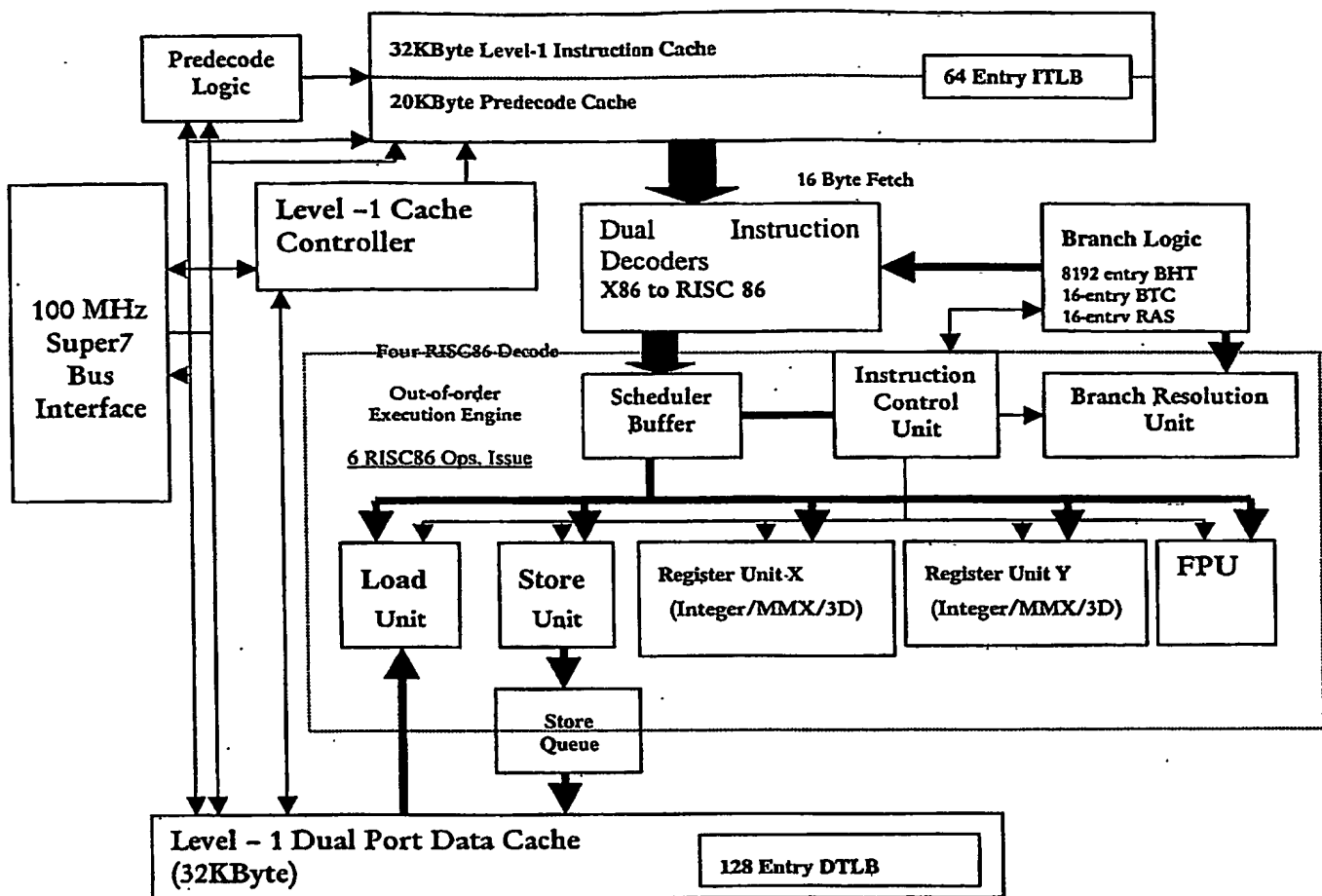


Figure 3 Generic architecture for a 3D NOW! Based AMD processor

The generic architecture of AMD processors with 3D Now enabled technology is shown in Figure 3. Details of the instruction set followed by the micro-architecture are as follows:

**Instruction Set features:**

- 21 instructions
- Support for SIMD floating-point and integer operations
- Specific SIMD integer instruction to enhance MPEG decoding
- New PREFETCH instruction to eliminate extra data retrieval time
- FEMMS (Fast Entry/Exit Multimedia State) instruction to reduce switching time between MMX and x87 code
- Open-standard support of IEEE 754 single precision data type

**Processor Micro-architecture features:**

- Fully pipelined dual execution resources
- Unlimited storage of floating-point numbers in memory
- Execution of up to two 3DNow! instructions per clock
- Total of four floating-point calculations (add, subtract, multiply) per clock (Enables potential peak performance of 1.2 Gigaflops at 300MHz vs. potential peak performance of 0.3 gigaflops for 300MHz processors without 3DNow! technology)
- Common floating-point stack; eliminates task switching between AMD-3Dnow! and MMX operations

The CISC processors available today are 32 bit architectures. A collaborative effort between Intel and HP has resulted in the definition of a new 64-bit architecture, viz. IA-64.

### IA-64 architecture

The Intel-HP architecture team designed IA-64 [47] to permit future expansion by providing sufficient architectural capacity, a full 64-bit address space, large directly accessible register files, enough instruction bits to communicate information from the compiler to the hardware, and the ability to express arbitrarily large amounts of ILP. IA-64 realizes parallel execution semantics in the form of instruction groups. The compiler creates instruction groups so that all instructions in an instruction group can be safely executed in parallel.

While instruction groups allow independent computational instructions to be placed together, expressing parallelism in computation related to program control flow requires additional support. Control parallelism is present when a program needs to select one of several possible branch targets, each of which might be controlled by a different conditional expression. Such cases would normally need a sequence of individual conditions and branches. IA-64 provides multi-way branches that allow several normal branches to be grouped together and executed in a single instruction group. The use of parallel compares and multi-way branches can substantially decrease the critical path related to control flow computation and branching.

While the compiler can handle some activities, hardware better manages many other areas including branch prediction, instruction caching, data caching, and prefetching.

For these cases, IA-64 improves on standard instruction sets by providing an extensive set of hints that the compiler uses to tell the hardware about likely branch behavior (taken or not taken, amount to prefetch at branch target) and memory operations (in what level of the memory hierarchy to cache data). The hardware can then manage these resources more effectively, using a combination of compiler-provided information and histories of runtime behavior. To help reduce the effect of branch mispredictions, IA-64 provides predication, a feature that allows the compiler to execute instructions from multiple conditional paths at the same time, and to eliminate the branches that could have caused mispredictions.

IA-64 provides a class of load instructions called speculative loads, which can safely be scheduled before one or more prior branches. In the block where the programmer originally placed the load, the compiler schedules a speculation check. In IA-64, this process is referred to as control speculation. IA-64 also has instructions that allow the

compiler to schedule a load before one or more prior stores, even when the compiler is not sure if the references overlap. This is called data speculation.

The IA-64 FP architecture is a unique combination of features targeted at graphical and scientific applications. It supports both high computation throughput and high-precision formats. The inclusion of integer and logical operations allows extra flexibility to manipulate FP numbers and use the FP functional units for complex integer operations. The primary computation workhorse of the FP architecture is the FMAC instruction, which computes a multiply accumulate operation with a single rounding. Traditional FP add and subtract operations are variants of this general instruction.

Divide and square root is supported using a sequence of FMAC instructions that produce correctly rounded results. Using primitives for divide and square root simplifies the hardware and allows overlapping with other operations. For example, a group of divides can be software pipelined to provide much higher throughput than a dedicated non-pipelined divider.

### 3.3 Comparison of General Purpose Processors

Table 1 compares the most salient features of some significant state of the art general-purpose microprocessors with specialized media extended instruction sets.

- The processors issue two or more instructions per cycle through the superscalar (SS) architectural feature. This enables multiple multimedia instruction to be issued and executed in parallel. The number of instruction decoders, which range from 1 to 4, indicates this feature. Pipelining (P) is also adopted to increase the efficiency of instruction issue and execution.
- These processors have out-of-order issue (OOI) control mechanism, which is supported by clock speeds in excess of 700 MHz. The out-of-order control unit requires additional functional units such as a reorder buffer that controls the instruction issue and a reservation station that reorders the actual instruction issues for the execution units and renamed register files. These components occupy a large portion of the silicon area and contribute to the power dissipation. These general-purpose processors have large power ratings, ranging from 12 watts to 74 watts.
- Although the superscalar control unit consumes a large amount of hardware real estate, this feature does not yet exploit parallelism in media processing to the fullest extent.
- These processors employ dynamic branch prediction technique. The advantage of this technique lies in the ability to predict branching operations and hence avoid fetching instructions from the main memory. But this implies the need for large primary and secondary level caches. This occupies valuable silicon area and proves disadvantageous during cache misses due to real time constraints that need to be met for media applications.
- The cache mechanism is designed to use one-dimensional locality of consecutive addresses, but media rich applications require multi-dimensional locality of accesses. To increase the probability of cache hits, these processors are equipped with high instruction depths (up to 72).
- The word-lengths of the next generation microprocessors such as the IA-64 based Itanium have increased from 32 to 64 bits. But the word lengths needed for

multimedia processing applications are 8, 16 or 24 bits, much shorter than the data paths of these processors. This drawback can be overcome to some extent by specialized pack instructions, but is not an optimized solution.

- They implement in excess of 15 million transistors to provide for the large area consuming features mentioned above.

Therefore conventional general-purpose programmable processors lack the judicious combination of sufficient processing muscle, low power consumption and low real estate realizations to support the ever increasing demand for mobile multimedia processing. Hence there arose a need for new high performance processors for multimedia applications. These special purpose programmable processors evolved from the conventional DSP [48] architectures to the state of the art Media processors.

Table 1 State of the art general purpose programmable processors

Processor	Clk MHz	Cache	Power	Transistors (million)	MM support	Dynamic Branch Prediction	# of instruction decoders	Instruction depth (#)	Floating point Exec units/ Mmx units/ SSE			
									#	SS	P	OOI
Duron (mobile)	700	L1: 128kb L2: 64kb	X	25	3Dnow	Y	3	72	3	Y	Y	Y
Athlon	1333	L1: 128kb L2: 256kb	73W	37	3Dnow	Y	3	72	3	Y	Y	Y
Pentium 111 (mobile)	900	L1: 32kb L2: 256kb	34W	28	SSE	Y	3	X	2/1/1	Y	Y	Y
Pentium 4	1700	L1: 8+12kb L2: 256kb	64W	42	SSE2	Y	1	X	2/1/1	Y	Y	Y
UltraSPAR C-III	900	L1: 64kb L2: 1Mb	70W	29	VIS	Y	4	X	2	Y	Y	Y

#### 4 Special Purpose programmable Processors:

The special purpose programmable processors exploit the redundancies involved in media processing algorithms through the use of multiple floating point and media specific execution units. They also extract parallelism that exists at various levels, viz.: data, instruction/ subword and thread.

Processors that extract parallelism at the data level are encompassed under the umbrella of SIMD based architectures.

##### 4.1 SIMD based architectures:

Typical operations encountered such as SAD require the same operation to be performed on multiple pixels or data units with no dependencies involved. Therefore the same instruction can be used to perform this task simultaneously on all the data units. Processors exploiting this feature include [27] and [49]. The MAP 1000A from equator technologies has a 64 bit Partitioned Unit (called Graphics unit for partitioned arithmetic) and a 128 bit Partitioned Unit (called Media Unit for SAD, inner product arithmetic) which exploit data level parallelism. The MPEG-4 processor proposed by [49] has a Video Unit containing 2 DCT units (DCT + Q), 2 ME units (Coarse and fine) and a MC unit. The Audio unit exploits this feature through an Inv Modified DCT unit.

The single-chip Video Audio Signal Processor (VASP) discussed by [50] consists of a video signal processing block. The video signal processing block contains a DCT/Q unit and 2 ME units designed to implement hardware solution of pixel input/output, full pixel motion estimation, half pixel motion estimation, discrete cosine transform and quantization. The D30V/MPEG [21] multimedia processor which supports real time MPEG-2 decoding has a 2 way SIMD multimedia core.

The 200-MHz embedded RISC processor for multimedia applications by [51] has a 64-bit SIMD based function unit that contributes a total of five multiply-adders in the processor. The unit is pipelined to 6 stages. It has a Multiply unit, Add unit, Shift unit, Logical function unit, 2 data type converter units and a 32 word 64 bit Register file. The Video Multi Processor (VMP) for image compression and decompression schemes of MPEG-2 proposed by [52] has a signal processor which is SIMD in nature and has 8 ALUs (16 bit each) and 8 MACs (16 bits each).

[53] discusses the architecture of the Multimedia Signal Processor with SIMD-type-parallel-executing features; byte-aligned-word-access features; and multi-instruction migrating features. The SH4 [54], the latest SH series microprocessor for multimedia applications from Sharp has architectural enhancement based on the unique floating-point vector instructions that are more effective than the conventional SIMD architecture for 3D graphics processing. Several other SIMD based processor were proposed by [55], [56].

Parallelism can also be exploited at the instruction level. The most popular approaches being VLIW, Super-Scalar and Super-pipelined architectures.

#### 4.2 VLIW processors:

VLIW processors use a long instruction word that usually contains a fixed number of operations that are fetched, decoded, issued and executed synchronously. All operations specified within a VLIW instruction must be independent of one another. Some key issues of a VLIW [57] processor are

- Very long instruction word (128 to 1024 bits per instruction)
- Each instruction consists of multiple independent parallel operations
- Each operation requires a statically known number of cycles to complete
- A central controller that issues a long instruction word every cycle
- Multiple FUs connected through a global shared register file.

The compiler groups independent instructions, executable in parallel, using optimization techniques such as software pipelining and loop unrolling and schedules code blocks. Therefore the VLIW processors cannot react to dynamic events such as cache misses.

Media processors exploit this level of parallelism by integrating VLIW based cores into their processing units. The Fujitsu FR500 embedded microprocessor, the first product in the FR-V line, Fujitsu's generic name for VLIW architecture microprocessors [23] offers a VLIW, 4 way, variable length Instruction Issue. Each instruction has a length of 32 bits. It supports an Instruction set consisting of Integer, Single precision floating point, media(fixed point) instructions. It has 2 (non pipelined) Integer execution units, 2 (2 stage pipelined) Floating point execution units and 4 (2 stage pipelined) Media execution units.

The MAPI000A [27] VLIW mediaprocessor is a single-chip, programmable media-processor that has a VLIW core with a 4 way Instruction Issue. The Instruction length is 34 bits (32 + 2 bit header)supporting an Instruction set of Integer, Single precision floating point, media(fixed point) instructions. It has 4 (non pipelined) Integer execution units, and the following Floating point execution units: 32 bit MAC- 2 (2 stage pipelined), 32 bit Divide/ Sqr root- 2 (2 stage pipelined), 64 bit Partitioned Unit (called Graphics unit for partitioned arithmetic) and a 128 bit Partitioned Unit (called Media Unit for SAD, inner product arithmetic).

M-PIRE [12] is a programmable MPEG-4 multimedia codec VLSI for mobile and stationary applications. It integrates a RISC core, two separate DSPs, a 64-bit dual-issue VLIW macroblock engine, and an autonomous I/O processor on a single chip to cope with the high flexibility and processing demands of the MPEG-4 standard. It supports real-time video and audio processing of MPEG-4 simple profile or ITU H.26x standards. The MacoBlock unit (part of the Video Unit) has a 2 way VLIW Instruction Issue and an Instruction length of 64 bits.

The D30V/MPEG multimedia processor [21] also has a 2 way VLIW instruction issue. The TANGRAM VLSI co-processor [28] intended as a building block for use in system-on-chip (SOC) designs for the versatile MPEG-4 multimedia standard is yet another example. It is designed to perform the computation intensive final step of MPEG-4 video decoding: compositing of scenes at the display. This includes warping and alpha blending of multiple full-screen video textures in real-time. TANGRAM consists of a 16-bit RISC control processor with a VLIW issue and multiple powerful arithmetic units that perform rendering calculations directly in hardware.

The Philips TM1300, a member of the Trimedia family [58], [59] of processors (Figure 6) contains an ultra-high performance VLIW processor, as well as a complete intelligent video and audio input/output subsystem. The processor has an instruction set that is optimized for processing audio, video and graphics. It also includes powerful SIMD multimedia operators for eight- and 16-bit signal data types as well as a full complement of 32-bit IEEE compatible floating point operations. TM1300 is intended as a multi-standard programmable video, audio and graphics processor. It can either be used standalone, or as an accelerator to a general-purpose processor.

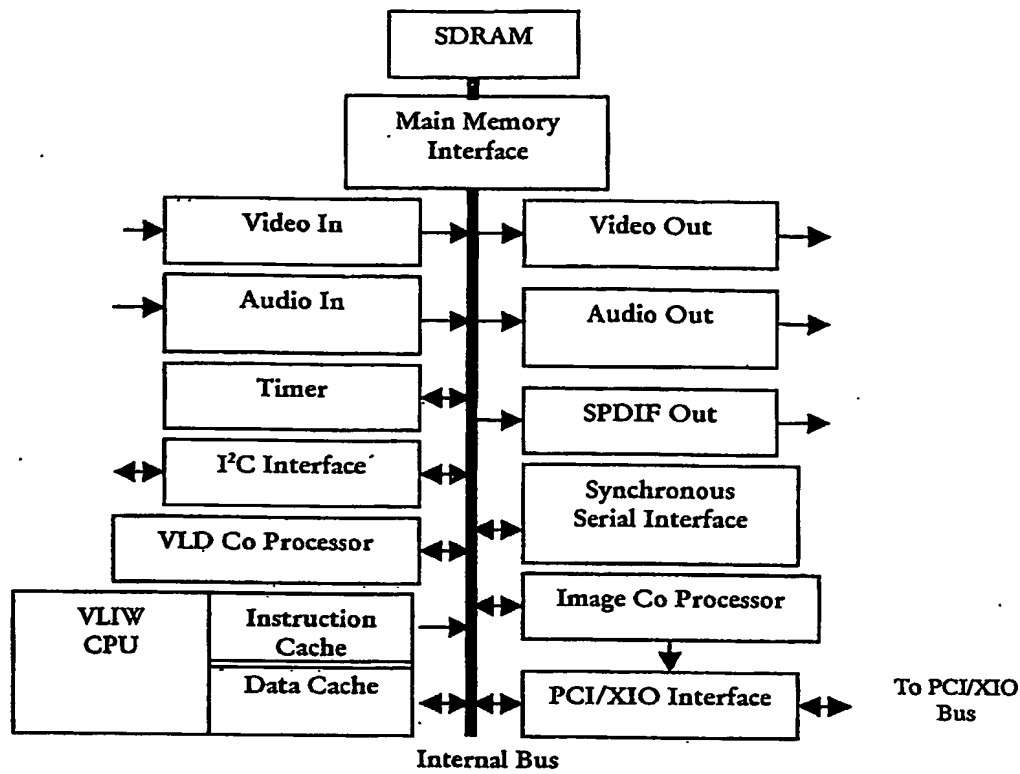


Figure 4 TM 1300 Block Diagram

The key features of TM1300 (figure 4) are

- A powerful, general-purpose VLIW processor core (the DSPCPU) that coordinates all on-chip activities. In addition to implementing the non-trivial parts of multimedia algorithms, this processor runs a small real-time operating system that is driven by interrupts from the other units.
- DMA-driven multimedia input/output units that operate independently and format data to increase the efficiency of software media processing.
- DMA-driven multimedia coprocessors that operate independently and in parallel with the DSPCPU to perform operations specific to important multimedia algorithms.
- A high-performance bus and memory system that provides communication between TM1300's processing units.
- A flexible external bus interface.

The TM1 has been used in a prototype version of the Microsoft Talisman processor architecture [60], [61]. The latest member of the Trimedia family is the TM32, which issues 5 instructions per clock, targeting 5 of the CPU's 27 functional units (FUs). It operates at a clock frequency of 250 MHz.

### 4.3 Thread control:

Speculative thread control is yet another level of parallelism that can be exploited. This concept is clearly exploited in Sun's MAJC architecture through the concepts of Space Time Computing and Vertical Multithreading.

#### (a) Space time computing:

Space Time Computing in the MAJC [62] architecture is a technique that substantially improves performance and code execution time in many applications using Java Technology. The multiprocessor-on-a-chip configuration in the MAJC architecture allows system level parallelism on a processor cluster to be achieved by having speculative threads (future instruction streams) execute on separate processors. For example, if we have two processors on a chip (Figure 5), then two threads —Head and Speculative —execute on separate processors. They operate in a different space (speculative heap) and in a different time (future time).

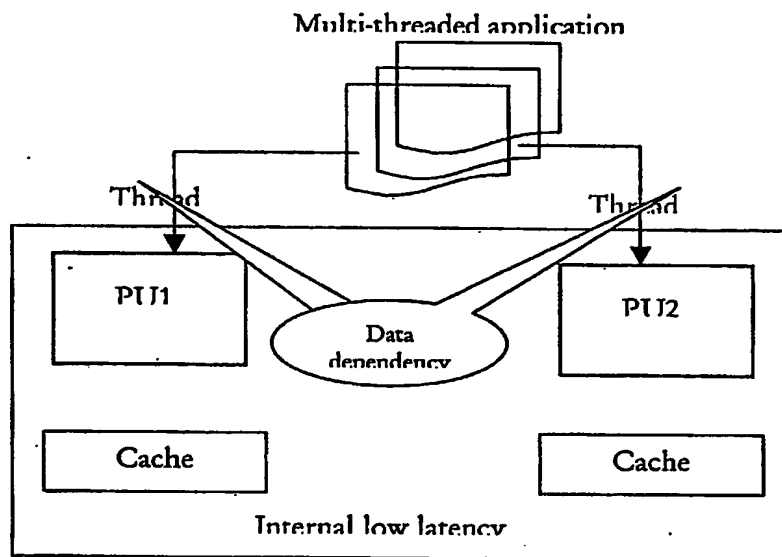


Figure 5 Space Time Computing

When programs have loops or method boundaries (as with Java), the MAJC architecture splits the program into instruction groups (threads) that are executed simultaneously on different processors as shown in the figure above. The first set of instructions or instruction group runs on CPU1 and is called the "head thread". The second instruction group executed on the CPU2 is called "speculative thread". The thread is called speculative because it executes the instruction group ahead of time. Thus program instructions that would be executed in "future time" have been made "current."

When program groups or methods are executed ahead of time, two aspects need careful attention:

- Speculative threads should not modify the processing or computations in the head thread.
- Speculative threads should always work with the coherent program order values.

Java technology helps differentiate between stores to heap and stores to stack at the bytecode level. In the Java programming language, one can differentiate between stores to heap and stores to stack at the bytecode level. This is a big advantage for Space Time Computing. It helps compilers in issuing special instructions that help to accurately determine possible rollbacks. Thus, fewer stores need to be monitored.

#### **(b) Vertical multithreading**

Vertical Multithreading is a technique where multiple threads operate on a cache miss within a processor unit and reduce the overall CPU idle time. It decreases the total CPU cycles required for program execution and increases throughput. The MAJC architecture substantially increases CPU utilization with vertical multithreading. A thread is a stream of instructions. In the simplest terms, vertical multithreading in the MAJC architecture allows the CPU to switch to a new instruction stream whenever there is a cache miss. This increase in CPU utilization, coupled with instruction level parallelism gained from VLIW, significantly improves overall throughput. In order for vertical multithreading to be effective, the load time from the memory to the cache (upon cache miss) should be much greater than the switch time between instruction streams. To enable fast context switching, references to the threads (states) need to be stored. The large register file in MAJC allows the architecture to maintain reference information of four threads and effect fast switching between instruction streams. The MAJC architecture permits monitoring and trapping any register overlaps that may occur when storing reference information of multiple threads. This is important to ensure accuracy in processing thread information.

#### **4.4 Comparison of Special Purpose Programmable Processors**

The advantage of realizing an algorithm on silicon is to exploit the speed offered by a direct implementation as compared to a state machine driven execution through a programmable model. In the programmable scenario, a data path that has already been realized on chip can execute only a specific set of operations. Any algorithm that does not have those specific operations needs to be interpreted in terms of those instructions. This consumes time and power.

- These processors do not employ complex control mechanisms such as out-of-order issue, hence they consume lesser power than the general purpose microprocessors. Moreover they operate at lower frequencies, which reduces the power consumption. Most of the programmable processors that utilize a truly programmable core have power consumptions beyond 4-5 watts (as shown in Table 2). Only those processors that use a programmable core for the purpose of minimally controlling the functional units on the chip consume lower than 2-3 watts.
- Similar to conventional DSPs they use non cache on-chip memories. Some of them such as the MAP1000A, have features such as TLBs which enable virtual memory addressing like the general purpose microprocessors. This feature increases the addressable memory space, but reduces the speed of access.
- Most of these processors have VLIW control rather than superscalar control. This provides higher performance with lesser control circuit complexity.
- To enable higher arithmetic performance for media specific applications, they employ SIMD style of instruction execution. Here, the media processors have more function

units than issue slots. They also have features to support operations on parallel-packed data, which exploits parallelism to a large extent in media processing. A few of them like the FR500 and MAJC employ speculative thread control to optimize branch prediction.

- Processors like MAJC have registers that are data agnostic. This reduces the control mechanisms for separately handling the storage of floating and fixed-point data. With this feature the processor lends itself to be naturally scalable.
- Some of the processors offer the ease of programming and debugging through the use of high-level language support, but this requires high optimization of the VLIW code.

The above-mentioned features offer the advantages of low power and area, time optimized implementations. But the design and debugging phases involved in designing such application specific programmable processors, tends to drive the development cost higher. Moreover, for mobile multimedia applications, it is desirable to have power consumptions in the order of a watt or less.

Table 2 State of the art special purpose programmable processors

Processor	Data parallelism	Instruction parallelism	Thread parallelism	Memory (On chip RAM,		Clock MHz	HLL	Power Watts
	SIMD	VLIW	Speculative control	Mem	TLB			
FR500		Y	Y	Y		266		
MAP1000A		Y		Y	Y	220	Y (c)	5
[9]	Y			Y		60		240m
M-PIRE	Y	Y		Y				
[16]	Y			Y		64		5
D30V	Y	Y		Y		300		2.5
VMP	Y			Y		50		5
TANGRAM		Y		Y		100		1

## 5 Dedicated Implementations:

The dedicated class of implementations competes and complements the programmable class of processors. In configurations where a central RISC controller coordinates the movement of data between various dedicated units, they act a complementary role. But in configurations where they are used as stand alone accelerators, they compete with the programmable class of processors. This class of processors can be classified into single chip/monolithic and distributed/chip set implementations. They have power dissipations in the order of 3-4 watts, but use less than 100K transistors per functional unit.

The C-cube DVxpress-MX25 codec [63] is a single-chip, multi-format codec with support for 25 Mbit/s DV formats (including DVCPRO, miniDV, and DVCam) and 4:2:2 MPEG-2. [64] discusses an ASIC for Motion Estimation and Compensation which can be used in conjunction with a fixed point DSP chip for MPEG-4 decoding. [12] discusses the design of a 2D discrete cosine transform (2D-DCT) processor. The processor has about 50k transistors. It is able to process 400 Mpixels per second and at a clock frequency of 600 MHz, which satisfies the requirements of real time high definition moving pictures in the MPEG-2 standard. Another DCT core processor has been proposed by [14]. This architecture applies a fast DCT algorithm and multiplier-accumulator based on the distributed algorithm, which reduces the hardware requirement and achieves high operational speeds.

[66] discusses a multiple chip based implementation, which proposes an array architecture for MPEG-4 image compositing. The coprocessor architecture works in parallel to an MPEG-4 video- and audio-decoder, and performs computation and bandwidth intensive low-level tasks for image compositing. The processor consists of an SIMD array of 16 DSPs to reach the required processing power for real-time image warping, alpha-blending and 3D rendering tasks. The flexible architecture allows the adaptation of the processing resources to the specific needs of different tasks and applications. The processor has an object-oriented cache architecture with 2D virtual address space, that allows concurrent and conflict-free access to shared data objects for all 16 DSPs.

From sections 3.3 and 4.4 and 5, we observe that the dedicated approach offers the advantages of high speed and low power, yet their design and debugging phases involve a significant amount of time thus being cost prohibitive making them unsuitable for low cost mobile media applications. The complexity, variety of techniques and tools, and the high computation, storage and I/O bandwidths associated with multimedia processing pose challenges, particularly from the points of scalability, resource utilization and real-time implementation. For example, compression standards such as MPEG-4 and JPEG 2000 that have been recently proposed, offer high interactivity to the user, which translates to a dynamic change in the computing resources at both the encoder and decoder units. From Tables 1 and 2 it can be observed that, although the programmable architectures offer flexibility in implementation, yet the power dissipation is not suitable for mobile applications. The power consumptions vary from 1 watt to 74 watts, which is an expensive solution for mobile applications demanding power consumptions below 500mW. Most of the processor architectures do not offer the facility to exploit parallelism at the thread level, which is essential to cater to flexibility at higher levels of the application as in the case of MPEG-4. Only a few of the special purpose programmable processors have high-level language programmability (which provides ease of programming and debugging); whereas most of them offer firmware-programming facility. These drawbacks have lead into the exploration of the reconfigurable architecture design space.

## 6 Need for Reconfigurable Media Processing

Multimedia processing involves the manipulation of various facets of media data that have a number of commonalities and differences with each other. The commonalities offer the potential to exploit the parallelism and redundancies, while the differences demand the use of a flexible approach. It is our belief that the recent MPEG-4 [67] standard has the complexity and flexibility that point to explore the opportunities for Reconfigurable computing. MPEG-4 employs a variety of algorithms and coding modes, requiring a more generic approach in hardware than the dedicated approach. Hence there is a need for:

- Assessment of resources required to implement MPEG-4 through a detailed complexity analysis [68], [69].
- Understand the static and dynamic nature of MPEG-4 operations with reference to size, quality and mode of operation.
- Design a dynamically configurable hardware for MPEG-4 implementation.

### 6.1 Complexity Analysis of MPEG-4 VVM

This paper analyzes the complexity of the VOP-related video profiles of the MPEG-4 standard in terms of basic operations. The computational complexity has been estimated from the perspectives of both the Encoder and Decoder. The MPEG-4 video data was classified at various levels of abstraction – Video Object Layer, Video Object Plane and Macroblock (MB). The complexity analysis was performed on each of the data types and estimated in terms of basic operations for the lowest level of classification. The overall complexity at any level of data classification can then be obtained by the aggregate of the complexities at the lower level. A probability model has been employed for a realistic estimate of the computational complexity. The encoding and decoding of the different kinds of VOPs (depending on the profile) that constitute an MPEG-4 video are now presented.

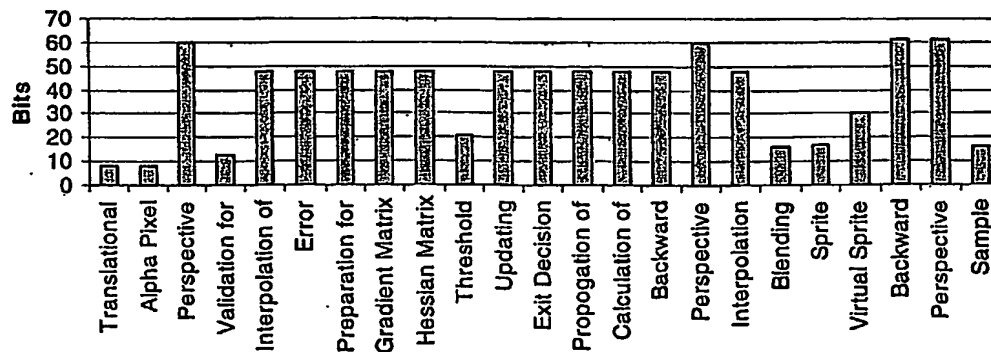
There are three kinds of VOPs, namely, the Intra (I), Predictive (P) and Bidirectionally Predictive (B) VOP. In order to analyze the complexity, all the possible coding modes for both the I-VOP and the P-VOP have been analyzed. An analysis of MPEG-4 video encoder and decoder tools shows that three classes of algorithms with similar properties exist. The grouping is as shown in Table 3. The analysis also shows significant differences in data widths and parallelism.

Table 3 Groups of algorithms within MPEG-4

Algorithm Type	Typical Data Width	Parallelism
Stream Processing	< 16 bit	Sequential, weak instruction level parallelism
Macroblock Processing	8, 16 bit	Data Parallelism
Graphic Processing	32 bit float	Data Parallelism, Task Parallelism

Based on the precision analysis in [13], we see that data width varies between 5 to 64 bits (Figure 6). In addition to integer operations, floating point operations are also a part of graphics processing. Figure 11 shows the large change in the complexity of the decoding process on a test sequence of a weather video in terms of MIPS.

Bit Precision for VOP Operations - Sprite



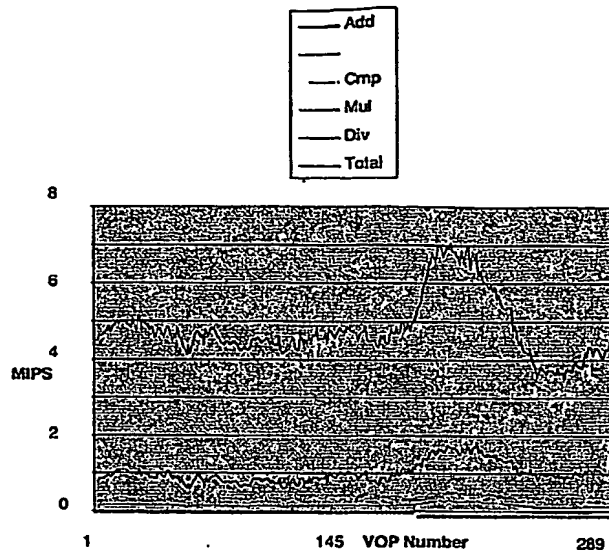


Figure 6 Complexity encountered in "Decoding" the Weather Sequence

The analysis of the Shape coding module of MPEG-4 is chosen as an example to demonstrate the potential for reconfigurability. The individual steps in the analysis process are as follows:

- Identification of the coding modes and the individual modules involved in shape coding.
- Identification of common resources among these modules.
- Number of times the resources are employed in each mode.

The operations involved in shape coding are as follows.

1. Motion Estimation
2. Motion Compensation
3. Mode Decision
4. Conversion ratio (CR) determination algorithm
5. Encoding Decision

The modes for encoding Intra BABs (Binary Alpha Block) are as shown in Figure 7.

The resources that are common to all operations in shape coding were identified as follows:

- Sum of Absolute Difference module (SAD)
- Down Sampler module (DS)
- UpSampler module (US)
- Transposer module (TR)
- Context based Arithmetic Encoder module (CAE)

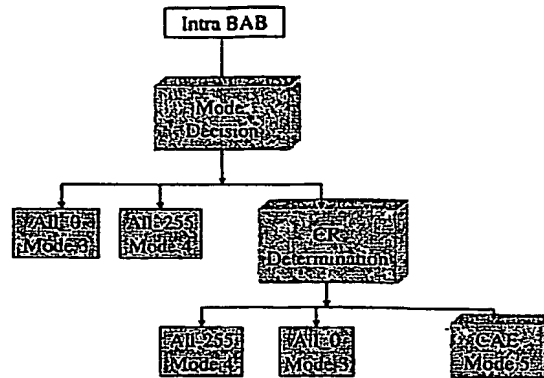


Figure 7 Encoding Algorithm for Intra BABs

From Figure 8 it can be observed that the resource SAD is extensively used in Modes 3, 4 and 5 of the Intra BAB encoding process, thus pointing to high reusability of these resources among the various modules.

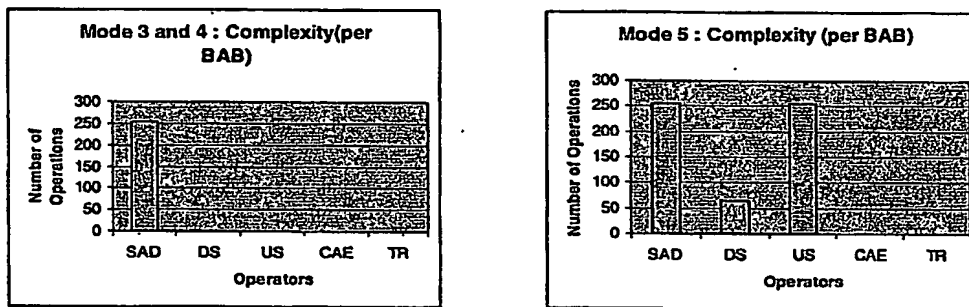


Figure 8 Complexity of Shape Coding of INTRA BABs

Table 4 shows the concurrency between the sub-operations involved in shape coding. It also shows the data dependency between the operations. This analysis aids in exploiting control parallelism between sub-operations. Each operation is associated with a unique number. For example, operation number 9 represents the upsampling of a shape block from 4x4 to 16x16 pixel size. As an example of concurrency, refer to the above table. Operation numbered 9 can concurrently be executed with operations 17,18,19,20 and 21.

Table 4 Concurrency between the sub-operations involved in shape coding

Shape coding	1	2	3	4	5	6	7	8	9	10	11	12
Motion Estimation	1											
Motion Compensation		2										
Mode Decision			3									
CR Determination												
* CR = 1/4												
1. Downsample from 16x16 to 4x4				7								
2. Upsample from 4x4 to 8x8					8							
3. Upsample from 8x8 to 16x16						9						
4. SAD							10					
* CR = 1/2												
1. Downsample from 16x16 to 8x8				4								
2. Upsample from 8x8 to 16x16					5							
3. SAD						6						
* CR = 1												
INTER CAE												
BAB encoding decision								11				

Motion Estimation ( Full Search)	1	2	3	4	5	6	7	8	9	10	11	12
Integer pixel motion estimation	12											
INTRA / INTER Mode Decision												
1. MB_mean		29										
2. Calculn of A		13										
3. Decision			14									
Half pixel Motion Estimation				15								
16 x 16 or 8 x 8 mode decision					16							
Differential coding of MVs												
1. Get Predictor						21						
2. Difference & MV Calculator						20						
Motion Compensation												
Bilinear Interpolation						19						
1. Normal Motion Compensation OR						18						
2. Overlapped Motion Compensation						17						
Normal DCT												
1. Field / Frame DCT decision							22					
2. DCT								23				
3. Saturation of coefficients									24			
4. Quantization of coefficients												
1. H.263 Quantization II										26		
2. Mpeg Quantization										25		
5. Prediction of coefficients											27	
6. AC prediction Enable decision												28

It can be seen that even in the case of shape coding module (which is a small portion of the entire MPEG-4 set of operations) there is a significant repetition in terms of the usage of the basic computational blocks. This points to the potential for extensive opportunities for exploiting reconfigurable implementations of multimedia algorithms. In the following section, we present an overview of reconfigurable solutions for general purpose computing followed by the proposed design methodology for designing a reconfigurable multimedia processor.

## 6.2 Reconfigurable Processors:

The availability of large, fast, FPGAs is making possible reconfigurable implementations (Figure 9). FPGAs consist of arrays of configurable logic blocks (CLBs) that implement various logical functions. The latest FPGAs from vendors like Xilinx and Altera can be partially configured in less than a millisecond. It is expected that devices with configuration times as low as 100 microseconds will be available within the next two years. Ultimately, computing devices may be able to adapt the underlying hardware dynamically in response to changes in the input data or processing environment.

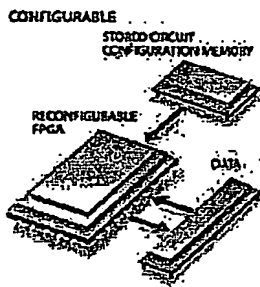


Figure 9 Configurable Computing

There are two dominant features that differentiate reconfigurable architectures [71] from special purpose computing architectures: (i) instructions which program the functionality of the device, and (ii) flexible interconnects which support task dependent data flow between operations. The reprogrammable space is characterized by the allocation and structure of these resources. Computational tasks can be implemented on a reconfigurable device with intermediate data flowing from the generating function to the receiving function. The salient features of reconfigurable machines are:

- Instructions are locally configured instead of broadcasting a new instruction every clock cycle, thus allowing the reconfigurable device to effectively process more instructions into active silicon in each cycle.
- Intermediate values are routed in parallel from producing functions to consuming functions (as space permits) rather than forcing all communication to take place through a central resource bottleneck.
- Memory and interconnect resources are distributed and are deployed based on need rather than being centralized, hence presenting opportunities to extract parallelism at various levels.

The networks connecting the CLBs can range from full connectivity crossbar to neighbor only connecting mesh networks. The best characterization to date which empirically

measures the growth in the interconnection requirements with respect to the number of look-up tables (LUTs) is the Rent's rule which is given as follows:

$$N^{io} = C N^p_{\text{gates}}$$

Where  $N^{io}$  corresponds to the number of interconnections (in/out lines) in a region containing  $N_{\text{gates}}$ .  $C$  and  $p$  are empirical constants. For logical functions typically  $p$  ranges from  $0.5 < p < 0.7$ .

It has been shown [71] (by building the FPGA based on Rent's model and using a hierarchical approach) that the instruction sizes in traditional FPGAs are higher than necessary, by at least a factor of 2-4. Therefore for rapid configuration, off-chip context loading becomes slow due to the large amount of configuration data that must be transferred across a limited bandwidth I/O path. It is also shown that greater word widths increase wiring requirements, while decreasing switching requirements. In addition, larger granularity data paths can be used to reduce instruction overheads. The utility of this optimization largely depends on the granularity of the data which needs to be processed. However, if the architectural granularity is larger than the task granularity, the device's computational power will be under utilized.

Some of the approaches towards designing reconfigurable processors for multimedia applications include PipeRench [72], MorphoSys [73] and Chimaera [74].

PipeRench is a coprocessor for streaming multimedia acceleration. PipeRench enables fast, robust compilers, supports forward compatibility, and virtualizes configurations, thus removing the fixed size constraint present in other fabrics. The PipeRench architecture has been optimized to balance the needs of the compiler against the limitations of variable data paths on silicon.

The MorphoSys reconfigurable system combines a reconfigurable array of processor cells with a RISC processor core and a high bandwidth memory interface unit. [75], [76] and [77] are other implementations that attempt to provide reconfigurable multimedia accelerator units.

CHIMAERA is a dynamically scheduled superscalar processor with a tightly-coupled reconfigurable functional unit. It is capable of: (1) mapping a set of instructions onto RFU operations, (2) converting control-flow into RFU operations.

Current approaches towards building a reconfigurable processor are targeted towards general purpose computing or a limited range of media specific applications and are not specifically tuned for mobile multimedia applications. The increasing demand for mobile multimedia processing applications with stringent constraints for low power, low chip area and high flexibility at both the encoder and decoder naturally demand the design and development of a dynamically reconfigurable multimedia processor. The methodology for the design of a reconfigurable media processor is now presented.

### 6.3 An Approach to building Reconfigurable Media Processors

A media processing algorithm coded in a high level language like C/ C++ or Java cannot effectively express the parallelism for execution on a hardware engine. In order to make a judicious decision on the relative components that need to be executed in hardware and software (running on a general purpose microprocessor), a suitable hardware software-partitioning algorithm must be designed which is targeted for media processing applications. This must be followed by scheduling the component that needs to be

implemented on hardware as well as temporal partitioning of the code onto the target architecture. The following steps summarize the proposed design philosophy.

**(a) Reconfigurable Module design**

A central feature in traditional engineering of computer systems is the requirement for partitioning of the system into hardware and software components [78]. Until recently, programmability was thought of only at the software level, but with the advent of Virtual Socket Interface (VSI) for system-on-chip design, there is also an element of programmability at the chip level. A common technique used for partitioning is to map the computationally expensive portions onto hardware and port the remaining portions onto software. One such approach [79] pre-synthesizes functions into hardware modules and stored as a library. We propose a tool to perform the partitioning and search for reconfigurable hardware patterns. This tool can be divided into 3 major modules:

- (i) **Parser and Recurring pattern analyzer**
  - (ii) **Parser Profiler**
  - (iii) **Partitioner and Router**
- **Parser and Recurring pattern analyser:** It generates a tree consisting of nodes and arcs to represent the data flow and control flow of the application. The input to the parser is the assembly code of the UltraSPARC V-9 architecture. The given C code is compiled using gcc with an optimization for the V-9 architecture. The need to adopt the analysis on the assembly code is that, it provides the advantages of
    - language independence
    - timing analysis for low-level granularity of operations such as parts of a functional equation.
    - It also gives statistics on the scheduling, number of registers used, number of memory transfers and timing of execution on a standard RISC processor against which proposed processor architectures can be compared.

The parser identifies (i) data flow graphs and (ii) control flow graphs.

The recurring pattern analyzer uses a clustering based approach to identify specific sequences of operations that can potentially be implemented in hardware. The clusters are sequences of fundamental RISC instructions. Based on a hierarchical approach, clustering results in modules with varying levels of granularity. The clusters are denoted by  $G_m^n$  where  $n$  denotes the granularity level and  $m$  denotes the type of cluster at the given level of granularity. A detailed analysis of the video encoding module of the MPEG-4 standard shows that parts of the assembly code to load indexed array members involve sequences of arithmetic computations along with data transfer operations which occur repeatedly over a large number of iterations and in almost all modules of the encoding process. A group of simple instructions such as load, move etc, form part of the lower of finer granular clusters with granularity level being one. Examples of such clusters are shown in Figure 10.

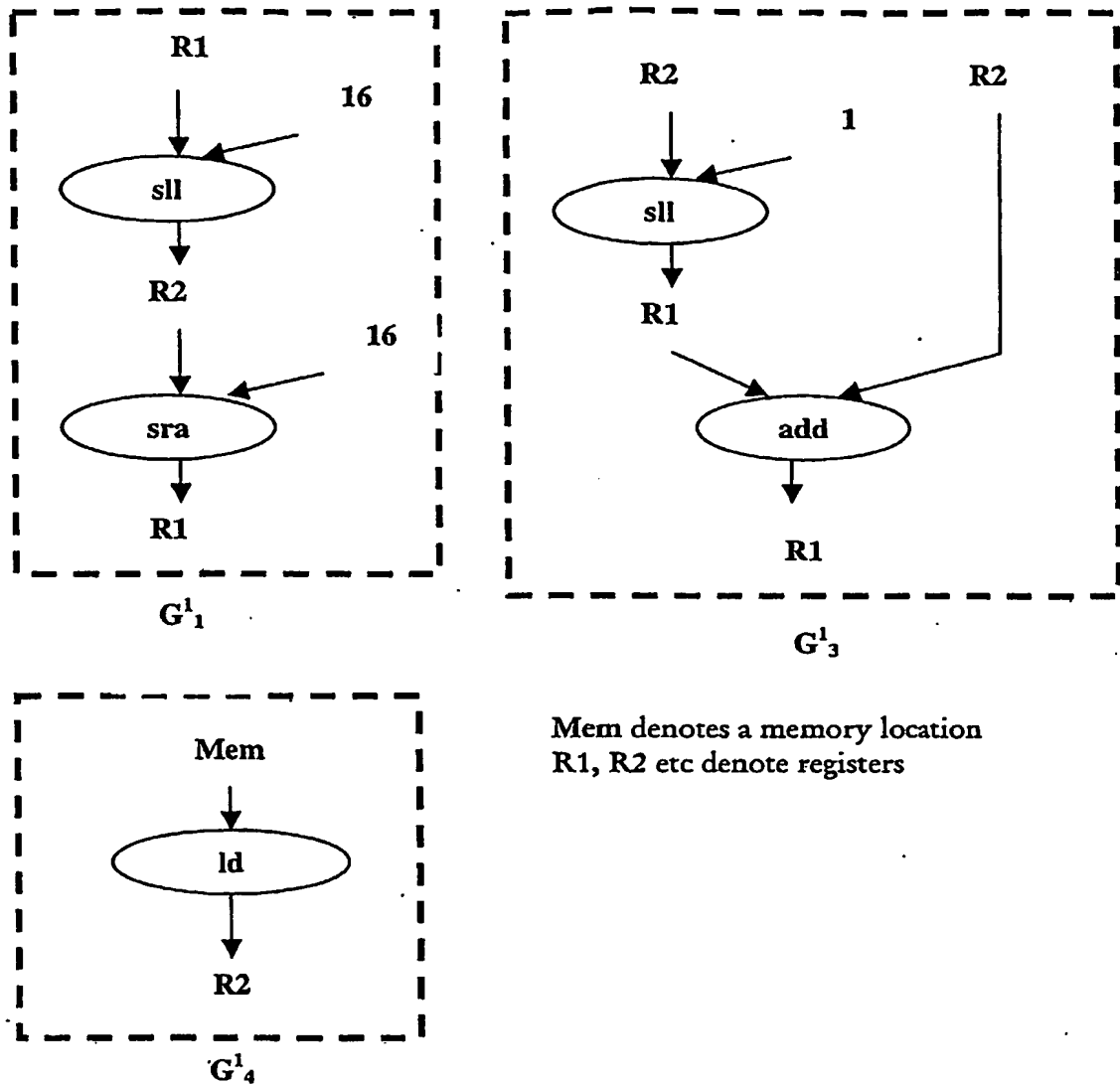


Figure 10 Level 1 Modules

At this level of granularity, the sources for the cluster opcodes are registers or memory locations or constants.

The following example illustrates the clustering process up to 4 levels of granularity. From the clustering mechanism, it can be observed that groups of low-level instructions can be grouped into modules which need not have any equivalent pattern at the functional level of the algorithm in the high level language. In Figure 11, a tree-based representation of the following computation is represented:

$$(y\_data[I-1] + y\_data[I+1] + y\_data[I+8])/3$$

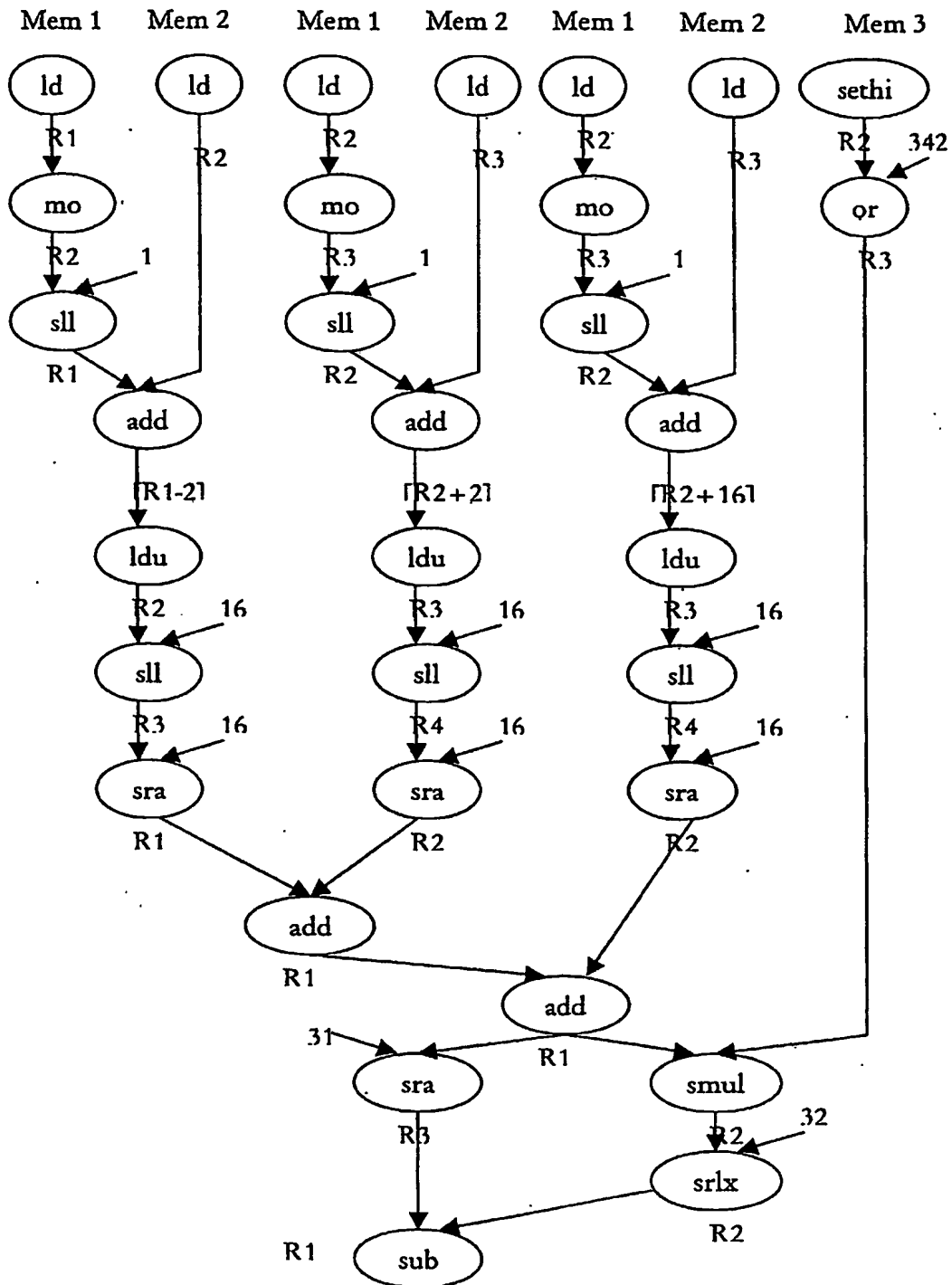


Figure 11 Tree based representation of a computation .

Figures 12 and 13 illustrate the clustering at level 1.

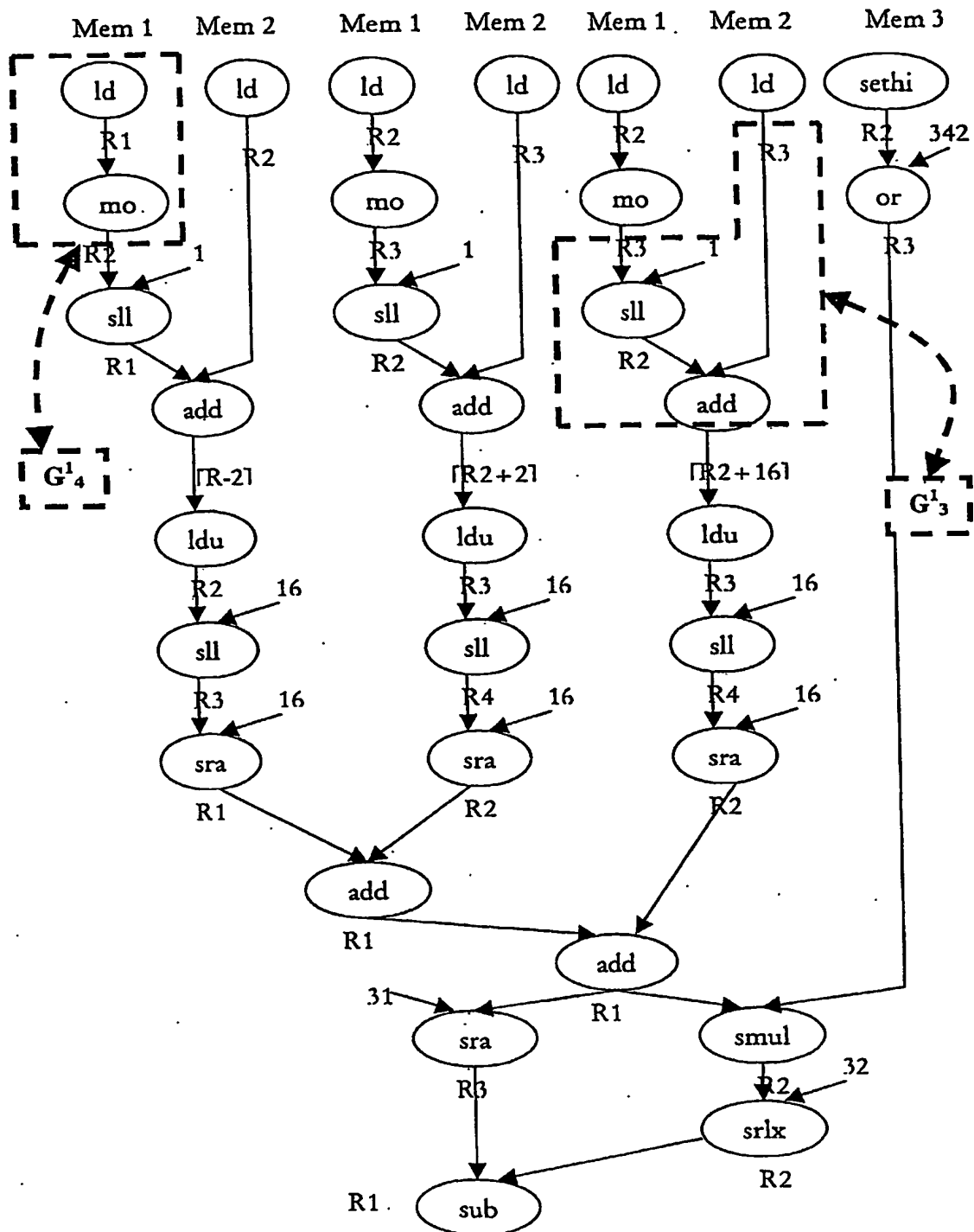


Figure 12 Replacement of instructions with level M modules

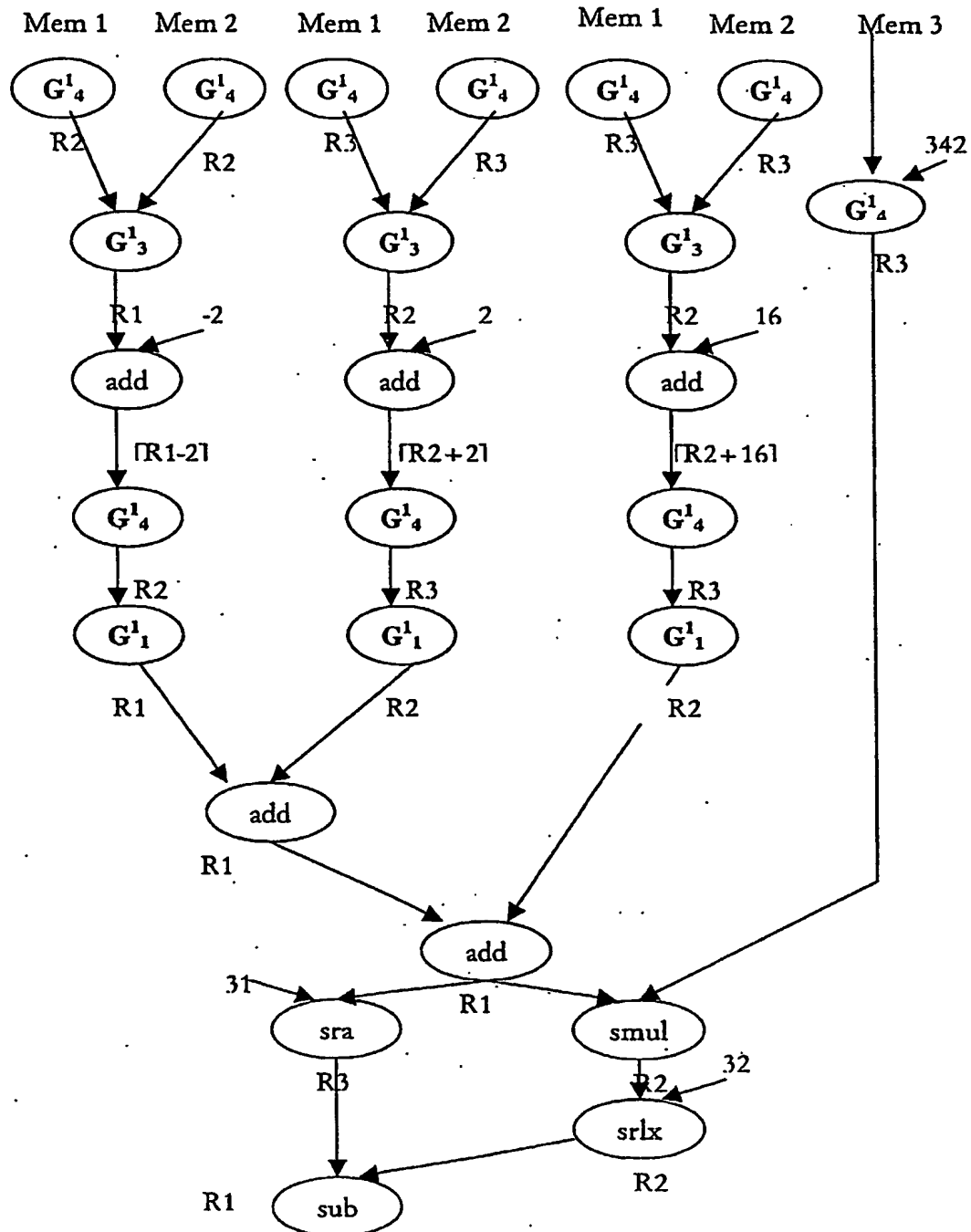


Figure 13 Tree with level M modules

A combination of sequences of level 1 and primary machine instructions, can be further clustered into a higher granularity cluster with level 2 and so on.

At the next higher level of granularity (level 2), the source opcodes are the level 1 modules. The destination opcodes can be arithmetic opcodes such as add, sub, mul etc. These constitute level 2 modules. Examples include  $G^2_1$ ,  $G^2_2$  etc shown in Figure 14:

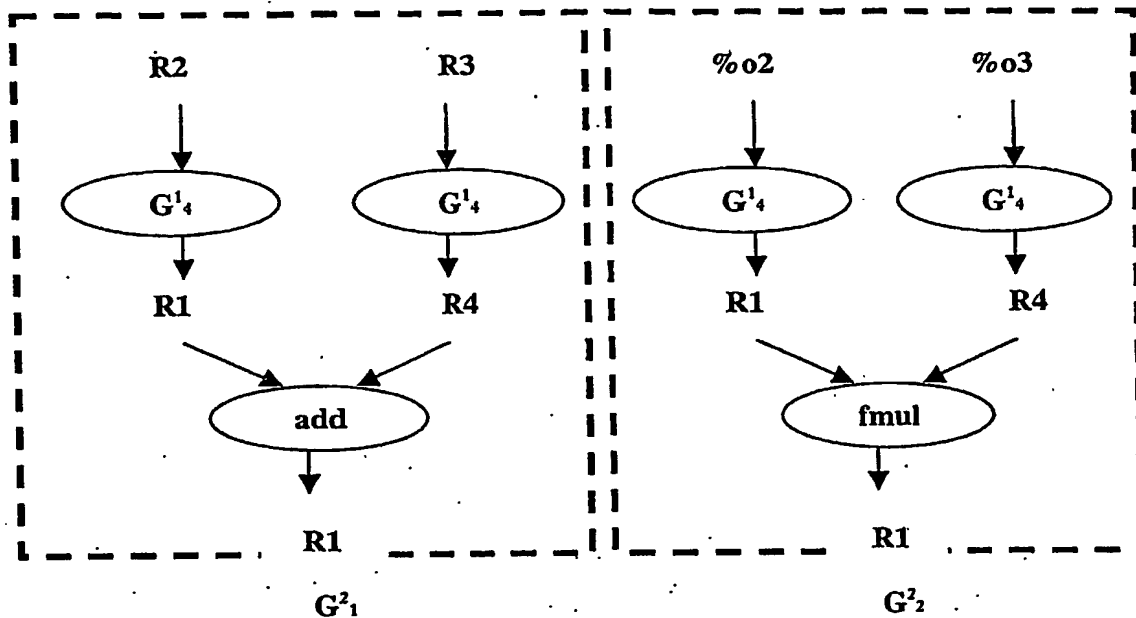


Figure 14 Level 2 Modules

The clustering tool searches all such combinations to obtain modules at higher levels of granularity. Some fundamental rules at higher levels of clustering include:

- (i) The destination opcode can be either an arithmetic operation or a module of the same level as the target cluster.
- (ii) Only two source opcodes can exist for one destination opcode.
- (iii) To cluster a pattern into a module of level  $n$ , at least one of the source opcodes should be of granularity  $n-1$  and none of them may be of the level  $n$  or higher.
- (iv) All modules are currently data agnostic.

The tree obtained from the profiler is scanned for recurring patterns in order to choose the optimal level of granularity of recurrence. These form the basic building blocks of the architecture.

In figure 15 and 16, coarser level of granularity based level 2 and 3 modules replace more complex instruction sequences.

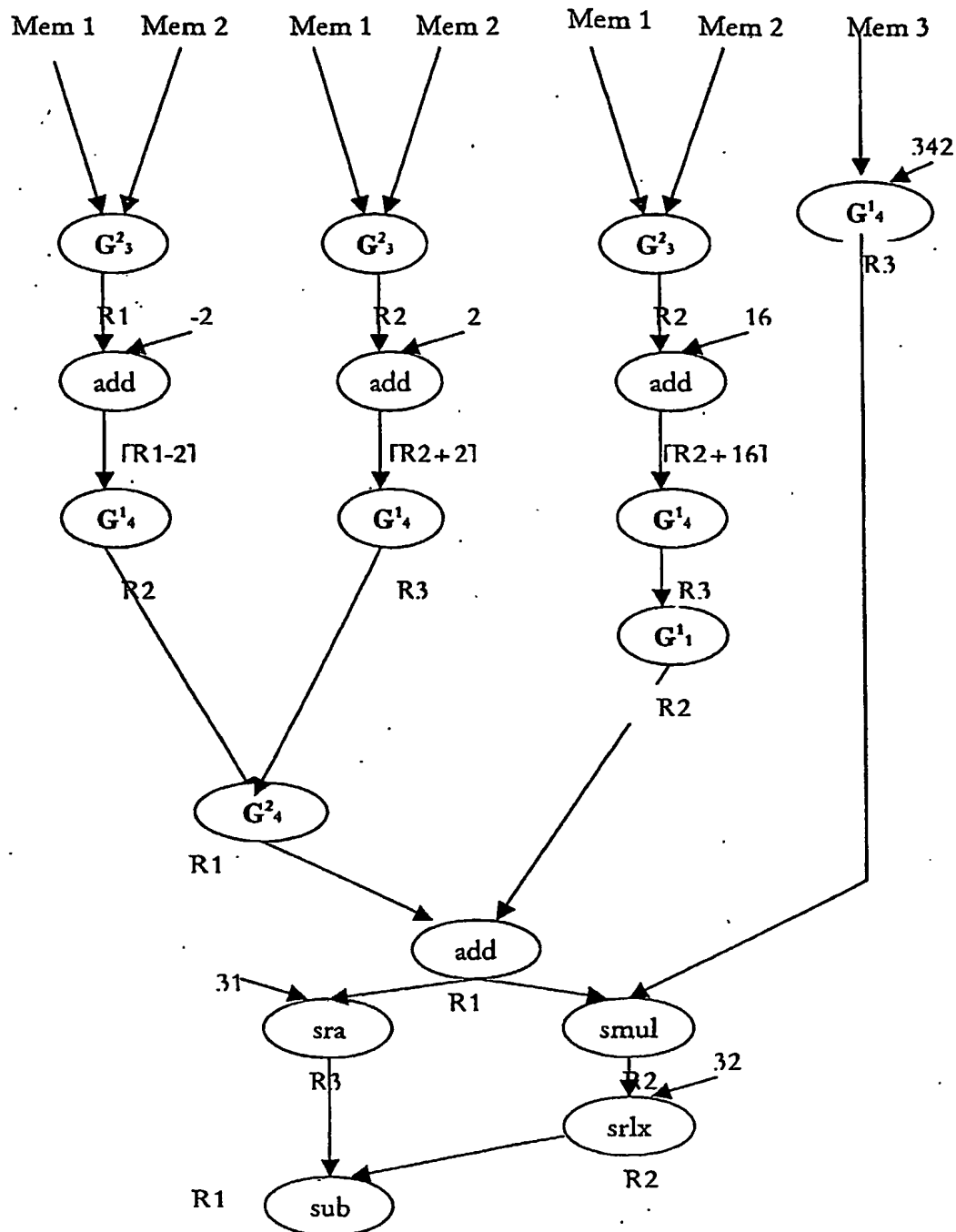


Figure 15 Tree with higher granularity level 2 and 1 modules

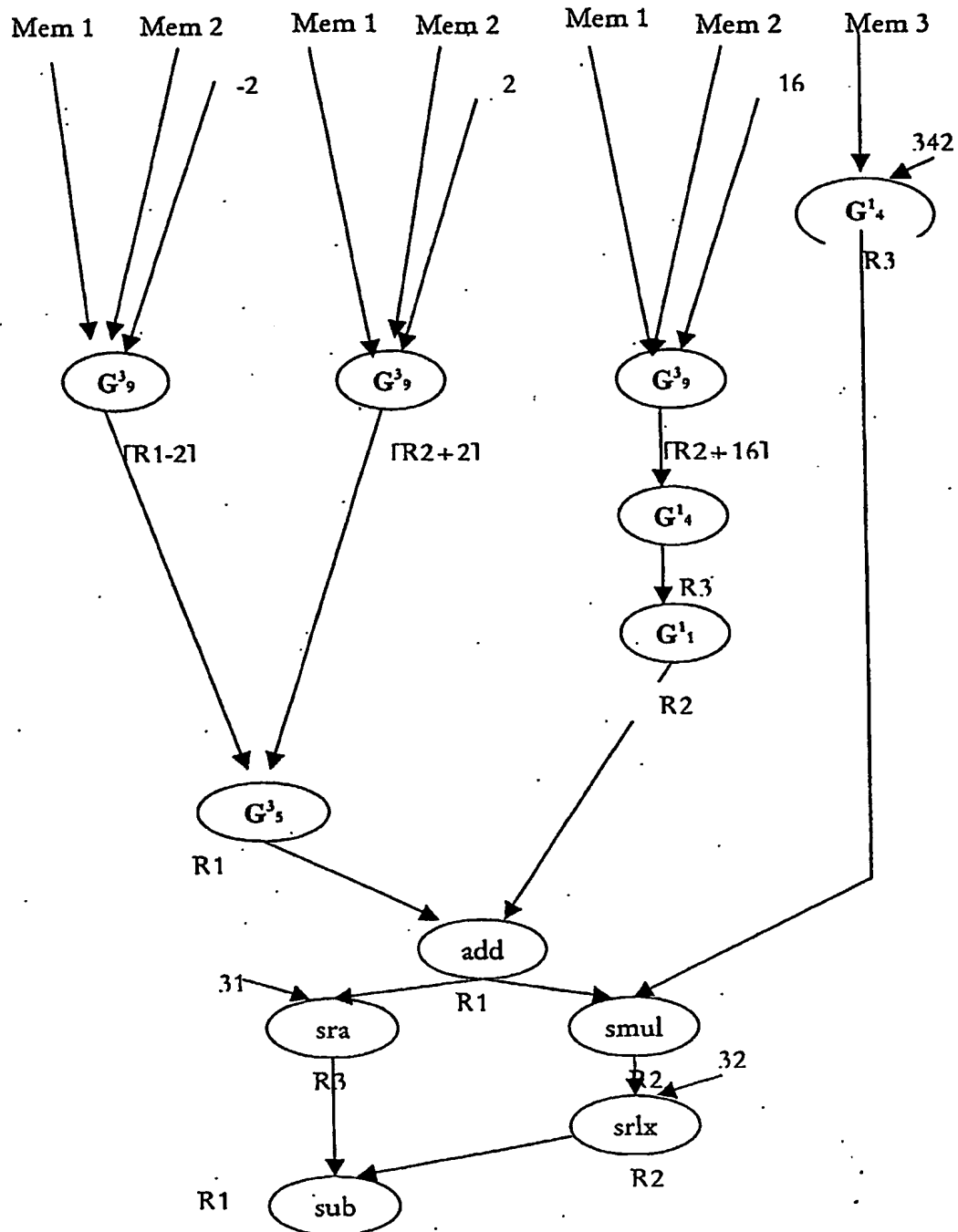


Figure 16 Tree with level 3 and 1 modules

A count of such modules at every level of granularity can be obtained for an entire algorithm. These statistics combined with the weights associated with each module based on the complexity of computation and data movement will help determine the extent of reconfiguration and the routing patterns on the processor. A library of hardware modules is developed for these patterns to facilitate the design of an embedded memory system, control logic and data paths of the reconfigurable processor.

The tool also performs an analysis of the control structures. Control flow graphs include structures such as for loops, do while loops and if-then-else statements. Detecting conditional structures such as if-then-else statements exposes thread level parallelism. Parallelism at the level of threads is a dominating factor in the design of hardware support for speculative execution. For example, the if-then-else control structure at the higher level of control such as

```
if(GetVopQuantType(curr))
{
    BlockQuantMPEG(coeff_ind, QP, Mode, type, qmat, qcoeff_ind)
}
else
{
    BlockQuantH263(coeff_ind, QP, Mode, type, qcoeff_ind)
}
```

launches two threads **BlockQuantMPEG** and **BlockQuantH263** in parallel. At any point during the quantization process if the feedback to the encoder necessitates a change from one type to another, then an instantaneous change can be applied due to the parallel running thread. A hierarchical based detection mechanism helps in restricting the search for cluster based patterns in data flow graphs in order to perform the search in real time. Currently we are designing algorithms to detect the patterns in real time.

- **Profiler:** It generates statistics about the execution behavior of the input application, e.g., the average number of times a loop is executed and the number of times a condition is successful. For an application like MPEG 4, the profiling tool presented in [80] is a good starting step. However, a more sophisticated tool based on the architecture of the underlying reconfigurable processor, needs to be developed to perform profiling for emerging multimedia standards. This takes into consideration both on-chip and off-chip memory transactions. Based on the profiling statistics,

additional weights are assigned to the nodes and arcs of the trees generated by the parser.

- **Partitioner and Router:** Based on the library built from the recurring pattern analysis and the feasibility of routing, the partitioner assigns modules of the input application to either hardware or software. The router places the hardware modules on the chip, in order to minimize the amount of data movement and maximize resource utilization between reconfigurations.

**(b) Temporal Partitioning:**

Temporal partitioning divides the design into mutually exclusive, limited size segments such that the logic required to implement a segment is less than or equal to the logic capacity of a configurable processor. Such segments can be scheduled for execution in proper order to ensure correct overall execution. Since run time reconfiguration imposes the constraint of changing resource count from cycle to cycle, there arises a need to perform a two layered scheduling before subjecting the hardware implementable modules to temporal partitioning. The first layer performs time constrained scheduling. The second layer performs a scheduling on the results of the first layer with either circular or learning/rule-based algorithms. The results of this process are then subjected to temporal partitioning.

A detailed complexity analysis of a variety of multimedia algorithms followed by the proposed methodology will enable the development of a fully automated hardware-software co-design tool. This will facilitate the design of dynamically reconfigurable multimedia processor.

## Conclusions

Multimedia processing is becoming increasingly important with wide variety of applications ranging from multimedia cell phones to high definition interactive television. Media processing involves the capture, storage, manipulation and transmission of multimedia objects such as text, handwritten data, audio objects, still images, 2D/3D graphics, animation and full-motion video. A number of implementation strategies have been proposed for processing multimedia data. These approaches can be broadly classified based on the evolution of processing architectures and the functionality of the processors. Based on evolution of the processing architectures, existing solutions for media processing can be broadly classified as Programmable processors, Dedicated implementations and Reconfigurable processors. Programmable processors include General Purpose Programmable processors based on a Complex Instruction Set or a Reduced Instruction Set and specialized programmable processors. The instruction sets of general-purpose programmable processors, which were originally meant only for general purpose applications, have now added media specific extensions to their Instruction Set Architectures (ISAs) to enhance the performance of media specific applications. Special purpose programmable processors have evolved starting with the early DSP processors (meant only for audio processing) to the state of the art Video/ Graphics processors. In order to provide media processing solutions to different consumer markets, designers have combined some of the classical features from both the functional and evolution

based classifications resulting in many hybrid solutions. We have also performed a detailed complexity analysis of the recent multimedia standard (MPEG-4) which has shown the potential for reconfigurable computing, that adapts the underlying hardware dynamically in response to changes in the input data or processing environment. We therefore propose a methodology for designing a reconfigurable media processor. This involves hardware-software co-design implemented in the form of a parser, profiler, recurring pattern analyzer, spatial and temporal partitioner. The proposed methodology enables efficient partitioning of resources for complex and time critical multimedia applications.

### References

1. "Model-Based Image Coding: Advanced Video Coding Techniques for Very Low Bit-Rate Applications", K. Aizawa and T. S. Huang, Proceedings of the IEEE, Vol. 83, No. 2, February 1995, pp. 259-271.
2. <http://www.transmeta.com/cruisoe/family.html>
3. <http://developer.intel.com/design/strong/sal10.htm>
4. The Microarchitecture of the Pentium® 4 Processor, Glenn Hinton, et.al Desktop Platforms Group, Intel Corp. Intel Technology Journal, First Quarter 2001.
5. AMD 3DNow! technology: architecture and implementations Oberman, S.; Favor, G.; Weber, F. IEEE Micro , Volume: 19 Issue: 2 , March-April 1999 Page(s): 37 -48
6. AMD's 3DNow!/sup TM/ vectorization for signal processing applications Dongho Kim; Gwangwoo Choe Acoustics, Speech, and Signal Processing, 1999. Proceedings., 1999 IEEE International Conference on , Volume: 4 , 1999 Page(s): 2127 -2130 vol.4
7. <http://www.amd.com/products/cpg/athlon/techdocs/pdf/22621.pdf>
8. UltraSparc I: a four-issue processor supporting multimedia Tremblay, M.; O'Connor, J.M. IEEE Micro , Volume: 16 Issue: 2 , April 1996 Page(s): 42 -50
9. AltiVec extension to PowerPC accelerates media processing Diefendorff, K.; Dubey, P.K.; Hochsprung, R.; Scale, H. IEEE Micro , Volume: 20 Issue: 2 , March-April 2000 Page(s): 85 -95.
10. <http://www.trimedia.com/products/briefs/cores.html>
11. The M-PIRE MPEG-4 codec DSP and its macroblock engine Stolberg, H.-J.; Berekovic, M.; Pirsch, P.; Runge, H.; Moller, H.; Kneip, J. Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on , Volume: 2 , 2000 Page(s): 192 -195 vol.2
12. A 600 MHz 2D-DCT processor for MPEG applications Sarmiento, R.; Pulido, C.; Tobajas, F.; Armas, V.; Esper-Chain, R.; Lopez, J.; Montiel Nelson, J.; Nunez, A. Signals, Systems & Computers, 1997. Conference Record of the Thirty-First Asilomar Conference on , Volume: 2 , 1997 Page(s): 1527 -1531 vol.2
13. MPEG-4 accelerator for PC based codec implementation Young-Kwong Lim; Jinsuk Kwak; Sanggyu Park Circuits and Systems, 1998. ISCAS '98. Proceedings of the 1998 IEEE International Symposium on , Volume: 4 , 1998 Page(s): 182 -185 vol.4
14. A 0.8  $\mu$ m / 100-MHz 2-D DCT core processor Yi-Feng Jang; JinnNang Kao; Jinn-Shiang Yang; Po-Chuang Huang Consumer Electronics, IEEE Transactions on , Volume: 40 Issue: 3 , Aug. 1994 Page(s): 703 -710

15. [http://www.merl.com/New\\_MERL/MERL-MH/MH-Projects/Receiver-Chip.htm](http://www.merl.com/New_MERL/MERL-MH/MH-Projects/Receiver-Chip.htm)
16. [http://www.altera.com/literature/ds/ds\\_ap2.pdf](http://www.altera.com/literature/ds/ds_ap2.pdf)
17. <http://www.xilinx.com/products/virtex/handbook/index.htm>
18. The ASIC and FPGA design challenge Darby, B. The Teaching of Digital Systems (Digest No. 1998/409), IEE Colloquium on , 1998 Page(s): 1/1 -1/3
19. "Reconfigurable Architectures got General-Purpose Computing", Andre DeHon. A.I Technical Report No. 1586. October 1996. Massachusetts Institute of Technology.
20. John V. Guttag, Communicating chameleons, Scientific American, July 1999
21. The D30V/MPEG multimedia processor Takata, H.; Watanabe, T.; Nakajima, T.; Takagaki, T.; Sato, H.; Mohri, A.; Yamada, A.; Kanamoto, T.; Matsuda, Y.; Iwade, S.; Horiba, Y. IEEE Micro , Volume: 19 Issue: 4 , July-Aug. 1999 Page(s): 38 -47
22. <http://www.sun.com/microelectronics/MAJC/documentation/>
23. Introducing the FR500 embedded microprocessor Suga, A.; Matsunami, K. IEEE Micro , Volume: 20 Issue: 4 , July-Aug. 2000 Page(s): 21 -27
24. Introducing the IA-64 architecture Huck, J.; Morris, D.; Ross, J.; Knies, A.; Mulder, H.; Zahir, R. IEEE Micro , Volume: 20 Issue: 5 , Sept.-Oct. 2000 Page(s): 12 -23.
25. [http://www.amd.com/products/cpg/athlon/architecture\\_wp.html](http://www.amd.com/products/cpg/athlon/architecture_wp.html)
26. The MAJC architecture: a synthesis of parallelism and scalability Tremblay, M.; Chan, J.; Chaudhry, S.; Conigliam, A.W.; Tse, S.S. IEEE Micro , Volume: 20 Issue: 6 , Nov.-Dec. 2000 Page(s): 12 -25.
27. The MAP1000A VLIM mediaprocessor Basoglu, C.; Woobin Lee; O'Donnell, J.S. IEEE Micro , Volume: 20 Issue: 2 , March-April 2000 Page(s): 48 -59
28. Architecture of an image rendering co-processor for MPEG-4 systems Berekovic, M.; Pirsch, P.; Selinger, T.; Wels, K.-I.; Miro, C.; Lafage, A.; Heer, C.; Ghigo, G. Application-Specific Systems, Architectures, and Processors, 2000. Proceedings. IEEE International Conference on , 2000 Page(s): 15 -24
29. Use IRAM for rasterization Kang, Y.; Torrellas, J.; Huang, T.S. Image Processing, 1998. ICIP 98. Proceedings. 1998 International Conference on , 1998 Page(s): 1010 -1013 vol.3
30. <http://developer.intel.com/design/chipsets/850/>
31. UltraSPARC-III: a 3rd generation 64 b SPARC microprocessor Lauterbach, G.; Greenley, D.; Ahmed, S.; Boffey, M.; Chamdani, J.; Si-En Chang; Chen, D.; Yu Fang; Holdbrook, K.; Hsieh, M.; Keish, B.; Melanson, R.; Narasimhaiah, C.; Petolino, J.; Tung Pham; Le Quach; Kit Tam; Duong Tong; Liuxi Yang; Kui Yau Solid-State Circuits Conference, 2000. Digest of Technical Papers. ISSCC. 2000 IEEE International , 2000 Page(s): 410 -411
32. UltraSPARC-III: designing third-generation 64-bit performance, Horel, T.; Lauterbach, G. IEEE Micro , Volume: 19 Issue: 3 , May-June 1999 Page(s): 73 -85
33. UltraSparc I: a four-issue processor supporting multimedia Tremblay, M.; O'Connor, J.M. IEEE Micro , Volume: 16 Issue: 2 , April 1996 Page(s): 42 -50
34. VIS speeds new media processing Tremblay, M.; O'Connor, J.M.; Narayanan, V.; Liang He IEEE Micro , Volume: 16 Issue: 4 , Aug. 1996 Page(s): 10 -20
35. An X86 microprocessor with multimedia extensions Draper, D.A.; Crowley, M.P.; Holst, J.; Favor, G.; Schoy, A.; Ben-Meir, A.; Trull, J.; Khanna, R.; Wendell, D.; Krishna, R.; Nolan, J.; Partovi, H.; Johnson, M.; Lee, T.; Mallick, D.; Frydel, G.; Vuong, A.; Yu, S.; Maley, R.; Kauffmann, B. Solid-State Circuits Conference, 1997.

- Digest of Technical Papers. 43rd ISSCC., 1997 IEEE International , 1997 Page(s): 172 -173, 450
36. Subword parallelism with MAX-2 Lee, R.B. IEEE Micro , Volume: 16 Issue: 4 , Aug. 1996 Page(s): 51 -59
  37. Multimedia enhanced general-purpose processors Wong, S.; Cotofana, S.; Vassiliadis, S. Multimedia and Expo, 2000. ICME 2000. 2000 IEEE International Conference on , Volume: 3 , 2000 Page(s): 1493 -1496 vol.3
  38. <http://www.motorola.com/SPS/PowerPC/Altivec/facts.html>
  39. 64-bit and multimedia extensions in the PA-RISC 2.0 architecture Lee, R.; Huck, J. Compton '96. 'Technologies for the Information Superhighway' Digest of Papers , 1996 Page(s): 152 -160
  40. <http://www.arm.com/armwww.ns4/html/TRMs?OpenDocument>
  41. A 2 V 250 MHz multimedia processor Yoshida, T.; Shimazu, Y.; Yamada, A.; Holmann, E.; Nakakimura, K.; Takata, H.; Kitao, M.; Kishi, T.; Kobayashi, H.; Sato, M.; Mohri, A.; Suzuki, K.; Ajioka, Y.; Higashitani, K. Solid-State Circuits Conference, 1997. Digest of Technical Papers. 43<sup>rd</sup> ISSCC., 1997 IEEE International , 1997 Page(s): 266 -267, 471
  42. 1.38 cm/sup 2/ 550 MHz microprocessor with multimedia extensions Jain, A.K.; Preston, R.P.; Bannon, P.J.; Bertone, M.S.; Blake-Campos, R.P.; Bouchard, G.A.; Brasili, D.S.; Carlson, D.A.; Castelino, R.W.; Clark, K.M.; Kobayashi, S.; Lilly, B.P.; Mehta, S.; Miller, B.S.; Mueller, R.O.; Olesin, A.; Saito, Y.; Yalala, V. Solid-State Circuits Conference, 1997. Digest of Technical Papers. 43rd ISSCC., 1997 IEEE International , 1997 Page(s): 174 -175, 451
  43. Internet Streaming SIMD Extensions Thakkur, S.; Huff, T. Computer, Volume: 32 Issue: 12 , Dec. 1999 Page(s): 26 -34
  44. Implementing streaming SIMD extensions on the Pentium III processor Raman, S.K.; Pentkovski, V.; Keshava, J. IEEE Micro , Volume: 20 Issue: 4 , July-Aug. 2000 Page(s): 47 -57
  45. The Microarchitecture of the Pentium® 4 Processor, Glenn Hinton, et.al Desktop Platforms Group, Intel Corp. Intel Technology Journal, First Quarter 2001.
  46. <http://www.amd.com/products/cpg/athlon/techdocs/pdf/22621.pdf>
  47. Introducing the IA-64 architecture Huck, J.; Morris, D.; Ross, J.; Knies, A.; Mulder, H.; Zahir, R. IEEE Micro , Volume: 20 Issue: 5 , Sept.-Oct. 2000 Page(s): 12 -23.
  48. A 2.2 GOPS video DSP with 2-RISC MIMD, 6-PE SIMD architecture for real-time MPEG2 video coding/decoding Iwata, E.; Seno, K.; Aikawa, M.; Ohki, M.; Yoshikawa, H.; Fukuzawa, Y.; Hanaki, H.; Nishibori, K.; Kondo, Y.; Takamuki, H.; Nagai, T.; Hasegawa, K.; Okuda, H.; Kumata, I.; Soneda, M.; Iwase, S.; Yamazaki, T. Solid-State Circuits Conference, 1997. Digest of Technical Papers. 43rd ISSCC., 1997 IEEE International , 1997 Page(s): 258 -259, 469
  49. A 60-MHz 240-mW MPEG-4 videophone LSI with 16-Mb embedded DRAM Takahashi, M.; Nishikawa, T.; Hamada, M.; Takayanagi, T.; Arakida, H.; Machida, N.; Yamamoto, H.; Fujiyoshi, T.; Ohashi, Y.; Yamagishi, O.; Samata, T.; Asano, A.; Terazawa, T.; Ohmori, K.; Watanabe, Y.; Nakamura, H.; Minami, S.; Kuroda, T.; Furuyama, T. Solid-State Circuits, IEEE Journal of , Volume: 35 Issue: 11 , Nov. 2000 Page(s): 1713 -1721

50. An area efficient video/audio codec for portable multimedia application Seongmo Park; Seongmin Kim; Kyeongjin Byeon; Jinjong Cha; Hanjin Cho Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on , Volume: 1 , 2000 Page(s): 595 -598 vol.1
51. A 2000-MOPS embedded RISC processor with a Rambus DRAM controller Suzuki, K.; Daito, M.; Inoue, T.; Nadehara, K.; Nomura, M.; Mizuno, M.; Iima, T.; Sato, S.; Fukuda, T.; Arai, T.; Kuroda, I.; Yamashina, M. Solid-State Circuits, IEEE Journal of , Volume: 34 Issue: 7 , July 1999 Page(s): 1010 -1021
52. High-speed and low-power real-time programmable video multi-processor for MPEG-2 multimedia chip on 0.6 /spl mu/m TLM CMOS technology Seung-Min Lee; Jin-Hong Chung; Lee, M.M.-O. Design Automation Conference, 1999. Proceedings of the ASP-DAC '99. Asia and South Pacific , 1999 Page(s): 201 -204 vol.1
53. A high performance DSP architecture "MSPM" for digital image processing using embedded DRAM ASIC technologies Yoshizawa, H.; Tsuruta, T.; Kumamoto, N.; Kurita, M. ASICs, 1999. AP-ASIC '99. The First IEEE Asia Pacific Conference on , 1999 Page(s): 413 -416
54. SH4 RISC multimedia microprocessor Arakawa, F.; Nishii, O.; Uchiyama, K.; Nakagawa, N. IEEE Micro , Volume: 18 Issue: 2 , March-April 1998 Page(s): 26 -34
55. A 1.2-W, 2.16-GOPS/720-MFLOPS embedded superscalar microprocessor for multimedia applications Kubosawa, H.; Takahashi, H.; Ando, S.; Asada, Y.; Asato, A.; Suga, A.; Kimura, M.; Higaki, N.; Miyake, H.; Sato, T.; Anbutsu, H.; Tsuda, T.; Yoshimura, T.; Amano, I.; Kai, M.; Mitarai, S. Solid-State Circuits, IEEE Journal of , Volume: 33 Issue: 11 , Nov. 1998 Page(s): 1640 -1648
56. A media processor for multimedia signal processing applications Holmann, E.; Yoshida, T.; Yamada, A.; Mohri, A. Signal Processing Systems, 1997. SIPS 97 - Design and Implementation., 1997 IEEE Workshop on , 1997 Page(s): 86 -96
57. A VLIW Architecture for a Trace Scheduling Compiler, Colwell R.P et, all. Proceedings of the 2<sup>nd</sup> international Conference on Architectural Support for Programming Languages and Operating Systems, October, Palo Alto. CA. 180-192.
58. TriMedia CPU64 design space exploration Hekstra, G.J.; La Hei, G.D.; Bingley, P.; Sijstermans, F.W. Computer Design, 1999. (ICCD '99) International Conference on , 1999 Page(s): 599 -606
59. Design space exploration for future TriMedia CPUs Sijstermans, F.; Pol, E.J.; Riemens, B.; Vissers, K.; Rathnam, S.; Slavenburg, G. Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on , Volume: 5 , 1998 Page(s): 3137 -3140 vol.5
60. J. Kajiya and J. Torborg, "Talisman: Commodity realtime 3D graphics for the PC," in Proc. Int. Conf. Computer Graphics and Interactive Techniques (SIGGRAPH), Aug. 1996.
61. Architecture and Design of a Talisman-Compatible Multimedia Processor. Santanu Dutta, Member, IEEE, Vijay Mehra, Weiwen (Vivian) Zhu, Deepak Singh, Marcel Janssens, Ramakrishna Vengalasetti, Boaz Ben-Nun, Pardha Pothana, Venkat Adusumilli, Nahid (Mansuripur) King, John Yen-Han Huang, Lie (Laura) Ling, Chris Nelson, Jai Bannur, and Sarah Wu. IEEE transactions on circuits and systems for video technology, vol. 9, no. 4, june 1999 565

62. Exploiting Java instruction/thread level parallelism with horizontal multithreading Watanabe, K.; Wanming Chu; Yamin Li Computer Systems Architecture Conference, 2001. ACSAC 2001. Proceedings. 6th Australasian , 2001 Page(s): 122 -129
63. [http://www.c-cube.com/product\\_display.fcfm?ProdID=36](http://www.c-cube.com/product_display.fcfm?ProdID=36)
64. MPEG-4 accelerator for PC based codec implementation Young-KwongLim; Jinsuk Kwak; Sanggyu Park Circuits and Systems, 1998. ISCAS '98. Proceedings of the 1998 IEEE International Symposium on , Volume: 4 , 1998 Page(s): 182 -185 vol.4
65. Chip-set for video display of multimedia information Jaspers, E.G.T.; de With, P.H.N. Consumer Electronics, IEEE Transactions on , Volume: 45 Issue:3 , Aug. 1999 Page(s): 706 -715
66. A flexible processor architecture for MPEG-4 image compositing Berekovic, M.; Frase, R.; Pirsch, P. Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on , Volume: 5 , 1998 Page(s): 3153 -3156 vol.5
67. ISO/IEC JTC1/SC29/WG11, "MPEG-4 video verification model version 11.0," Doc. 2172, Mar. 1998.
68. "Complexity Analysis of MPEG-4 Video Profiles", A Master's thesis by C.N. Raghavendra. Arizona State University, 2000.
69. "Algorithms, Complexity Analysis and VLSI Architectures for MPEG 4 Motion Estimation", Peter Kuhn. Kluwer academic publishers.
70. "A methodology to Implement MPEG-4 Algorithms on reconfigurable Architectures", A Master's thesis by K. Ramaswamy. Arizona State University, 2000.
71. "Reconfigurable Architectures got General-Purpose Computing", Andre DeHon. A.I Technical Report No. 1586. October 1996. Massachusetts Institute of Technology.
72. PipeRench: a coprocessor for streaming multimedia acceleration Goldstein, S.C.; Schmit, H.; Moe, M.; Budiu, M.; Cadambi, S.; Taylor, R.R.; Laufer, R. Computer Architecture, 1999. Proceedings of the 26th International Symposium on , 1999 Page(s): 28 -39
73. MorphoSys: a reconfigurable architecture for multimedia applications Singh, H.; Ming-Hau Lee; Guangming Lu; Kurdahi, F.J.; Bagherzadeh, N.;Filho, E.M.C. Integrated Circuit Design, 1998. Proceedings. XI Brazilian Symposium on , 1998 Page(s): 134 -139
74. CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit Ye, Z.A.; Moshovos, A.; Hauck, S.; Banerjee, P. Computer Architecture, 2000. Proceedings of the 27th International Symposium on , 2000 Page(s): 225 -235
75. Integration of high-performance ASICs into reconfigurable systems providing additional multimedia functionality Blume, H.; Bluthgen, H.-M.; Henning, C.; Osterloh, P. Application-Specific Systems, Architectures, and Processors, 2000. Proceedings. IEEE International Conference on , 2000 Page(s): 66 -75
76. An application-tailored dynamically reconfigurable hardware architecture for digital baseband processing Becker, J.; Pionteck, T.; Glesner, M. Integrated Circuits and Systems Design, 2000. Proceedings. 13th Symposium on , 2000 Page(s): 341 -346
77. Coarse reconfigurable multimedia unit extension Wong, S.; Cotofana, S.; Vassiliadis, S. Parallel and Distributed Processing, 2001. Proceedings. Ninth Euromicro Workshop on , 2001 Page(s): 235 -242

78. "Tooling Up for Reconfigurable System Design", Gordon Brebner. 1999 IEE.
79. "Speeding up Program Execution Using Reconfigurable Hardware and a Hardware Function Library", Sitanshu Jain, M. Balakrishnan, Anshul Kumar, Shashi Kumar. 1997 IEEE.
80. "Algorithms, Complexity Analysis and VLSI Architectures for MPEG 4 Motion Estimation", Peter Kuhn. Kluwer academic publishers.

## Pattern Recognition Tool to Detect Reconfigurable Patterns in MPEG4 Video Processing

Ali Akoglu, Aravind Dasu, Arvind Sudarsanam, Mayur Srinivasan,  
 Sethuraman Panchanathan *Fellow IEEE*  
 Visual Computing and Communications Laboratory  
 Arizona State University  
 {ali.akoglu, dasu, arvind.sudarsanam, mayur.s, panch}@asu.edu

### Abstract

*Current approaches towards building a reconfigurable processor are targeted towards general purpose computing or a limited range of media specific applications and are not specifically tuned for mobile multimedia applications. The increasing demand for mobile multimedia processing with stringent constraints for low power, low chip area and high flexibility at both the encoder and decoder naturally demand the design and development of a dynamically reconfigurable multimedia processor. We have performed a detailed complexity analysis of the MPEG-4 video coding mode which has illustrated the potential for reconfigurable computing. We have recently proposed a methodology for designing a reconfigurable media processor. This involves the design of a parser that identifies data/control flow graphs generated from the input assembly code of an UltraSPARC V-9 architecture; recurring pattern analyzer that uses a clustering based approach to identify specific sequences of operations that can potentially be implemented in hardware; and finally a count of such modules at every level of granularity with the associated weights based on the complexity of computation and data transfers used by partitioner and router. In this paper we then propose the design of the parser and pattern recognizer with results for detecting the reconfigurable patterns in MPEG4.*

to implement MPEG-4 through a detailed complexity analysis [3], [4]; understanding the static and dynamic nature of MPEG-4 operations with reference to size, quality and mode of operation and finally designing a dynamically configurable hardware for MPEG-4 implementation.

A number of implementation strategies have been proposed for processing multimedia data. These approaches can be broadly classified based on evolution of media processing architectures and functionality. In order to provide media processing solutions to different consumer markets, designers have combined some of the classical features from both the functional and evolution based classifications resulting in many hybrid solutions. The complexity, real time constraints and the need for low power, area and cost efficient implementations can not all be satisfied by the existing solution strategies, therefore we have recently proposed a methodology for designing a reconfigurable media processor[1]. This involves hardware-software co-design implemented in the form of a parser, recurring pattern analyzer, partitioner and router. In this paper we propose the design of the parser and pattern recognizer with results. Paper is organized as follows. Section 2 gives an overview of reconfigurable processors, section 3 explains the methodology as well as the details of parser and pattern analyzer followed by results and conclusion.

## 1 Introduction

Multimedia processing involves the manipulation of various facets of media data that have a number of commonalities and differences with each other. The commonalities offer the potential to exploit the parallelism and redundancies, while the differences demand the use of a flexible approach. It is our belief that the recent MPEG-4 [2] standard has the complexity and flexibility that point to explore the opportunities for reconfigurable computing. Hence there is a need for assessment of resources required

## 2 Reconfigurable Processors:

The availability of large, fast, FPGAs is making reconfigurable implementations possible. FPGAs consist of arrays of configurable logic blocks (CLBs) that implements various logical functions. The latest FPGAs from vendors like Xilinx and Altera can be partially configured in less than a millisecond. It is expected that devices with configuration times as low as 100 microseconds will be available within the next two years. There are two dominant features that differentiate

reconfigurable architectures [5] from special purpose computing architectures: (i) instructions which program the functionality of the device, and (ii) flexible interconnects which support task dependent data flow between operations. The reprogrammable space is characterized by the allocation and structure of these resources. Computational tasks can be implemented on a reconfigurable device with intermediate data flowing from the generating function to the receiving function. The salient features of reconfigurable machines are:

- Instructions are locally configured instead of broadcasting a new instruction every clock cycle, thus allowing the reconfigurable device to effectively process more instructions into active silicon in each cycle.
- Intermediate values are routed in parallel from producing functions to consuming functions (as space permits) rather than forcing all communication to take place through a central resource bottleneck.
- Memory and interconnect resources are distributed and are deployed based on need rather than being centralized, hence presenting opportunities to extract parallelism at various levels.

Some of the approaches towards designing reconfigurable processors for multimedia applications include PipeRench [6], MorphoSys [7] and Chimaera [8]. PipeRench is a coprocessor for streaming multimedia acceleration. PipeRench enables fast, robust compilers, supports forward compatibility, and virtualizes configurations, thus removing the fixed size constraint present in other fabrics. The PipeRench architecture has been optimized to balance the needs of the compiler against the limitations of variable data paths on silicon. The MorphoSys reconfigurable system combines a reconfigurable array of processor cells with a RISC processor core and a high bandwidth memory interface unit. [9], [10] and [11] are other implementations that attempt to provide reconfigurable multimedia accelerator units. Chimaera is a dynamically scheduled superscalar processor with a tightly-coupled reconfigurable functional unit. It is capable of: (i) mapping a set of instructions onto reconfigurable functional unit (RFU) operations, (ii) converting control-flow into RFU operations. Current approaches towards building a reconfigurable processor are targeted towards general purpose computing or a limited range of media specific applications and are not specifically tuned for mobile multimedia applications. The methodology for the design of a reconfigurable media processor is now presented.

### 3 Parser and Pattern Analyzer

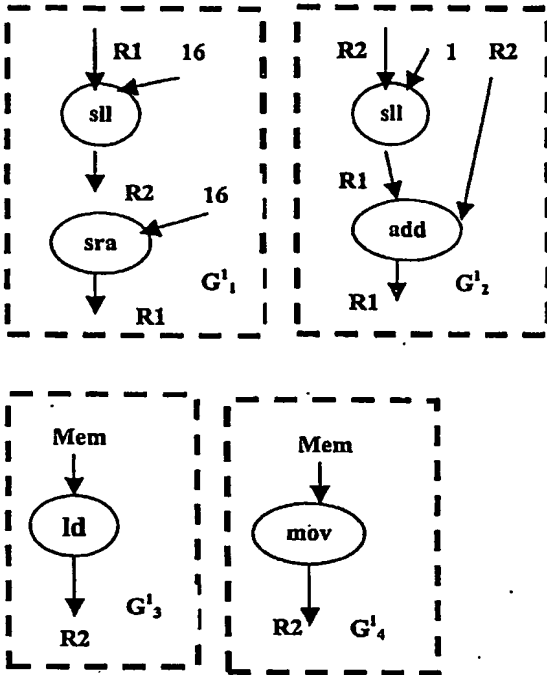
A media processing algorithm coded in a high level language like C/ C++ or Java cannot effectively express the parallelism for execution on a hardware engine. A

central feature in traditional engineering of computer systems is the requirement for partitioning of the system into hardware and software components [12]. A common technique used for partitioning is to map the computationally expensive portions onto hardware and port the remaining portions onto software. One such approach [13] pre-synthesizes functions into hardware modules and stored as a library. In order to make a judicious decision on the relative components that need to be executed in hardware and software (running on a general purpose microprocessor), a suitable hardware software- partitioning algorithm must be designed which is targeted for media processing applications. Hardware software partitioning must be followed by scheduling the component that needs to be implemented on hardware as well as temporal partitioning of the code onto the target architecture. We have recently proposed a methodology for the design of reconfigurable multimedia processor. The following steps summarize the proposed design philosophy perform the partitioning and search for reconfigurable hardware patterns.

- |       |                            |
|-------|----------------------------|
| (i)   | Parser                     |
| (ii)  | Recurring pattern analyzer |
| (iii) | Partitioner and Router     |

Parser generates a tree consisting of nodes and arcs to represent the data flow and control flow of the application. The input to the parser is the assembly code of the UltraSPARC V-9 architecture. The given C code is compiled using gcc with an optimization for the V-9 architecture. The need to adopt the analysis on the assembly code is that, it provides the advantages of language independence and timing analysis for low-level granularity of operations such as parts of a functional equation. It also provides statistics on scheduling, number of registers used, number of memory transfers and timing of execution on a standard RISC processor against which proposed processor architectures can be compared. The parser identifies (i) data flow graphs and (ii) control flow graphs. We propose the design of parser and reconfigurable pattern analyzer tool for detecting reconfigurable patterns in MPEG4. The recurring pattern analyzer uses a clustering based approach to identify specific sequences of operations that can potentially be implemented in hardware. The clusters are sequences of fundamental RISC instructions. Based on a hierarchical approach, clustering results in modules with varying levels of granularity. The clusters are denoted by  $G_m^n$  where  $n$  denotes the granularity level and  $m$  denotes the type of cluster at the given level of granularity. A detailed analysis of the video encoding module of the MPEG-4 standard shows that parts of the assembly code to load indexed array elements involve sequences of arithmetic computations along with data transfer operations which occur repeatedly

over a large number of iterations and in almost all modules of the encoding process. A group of simple instructions such as load, move etc, are typically present at the bottom level of finer granular clusters (granularity level being one). Examples shown in Figure 1.



Mem denotes a memory location  
R1, R2 etc denote registers

Figure 1 Level 1 Modules

At this level of granularity, the sources for the cluster opcodes are registers or memory locations or constants. The following example illustrates the clustering process up to four levels of granularity. From the clustering mechanism, it can be observed that groups of low-level instructions can be grouped into modules which need not have any equivalent pattern at the functional level of the algorithm in the high level language. In Figure 2 and 3, a tree-based representation of the following computation is represented:

$$(y\_data[I-1] + y\_data[I+1] + y\_data[I+8])/3$$

A combination of sequences of level 1 and primary machine instructions, can be further clustered into a higher granularity cluster (level 2) and so on. At the next higher

level of granularity (level 2), the source opcodes are the level 1 modules. The destination opcodes can be arithmetic opcodes such as add, sub, mul etc. These constitute level 2 modules. Examples include  $G^1$ ,  $G^2$  etc shown in Figure 4:

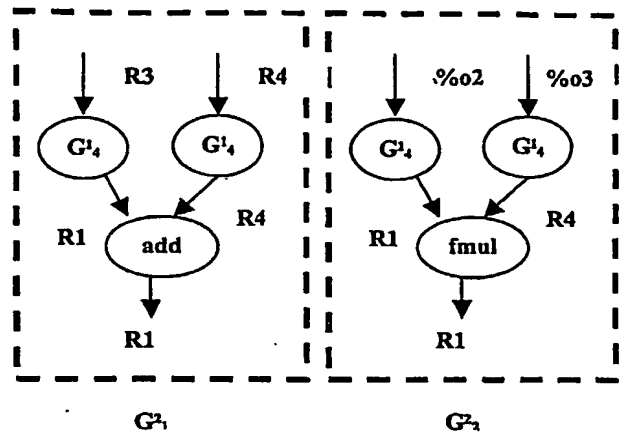


Figure 4 Level 2 Modules

The clustering tool searches all such combinations to obtain modules at higher levels of granularity. There are fundamental rules at higher levels of clustering such as the destination opcode can be either an arithmetic operation or a module of the same level as the target cluster. Only two source opcodes can exist corresponding to each destination opcode. To cluster a pattern into a module of level  $n$ , at least one of the source opcodes should be of granularity  $n-1$  and none of them may be of the level  $n$  or higher. The tree obtained from the parser is scanned for recurring patterns in order to choose the optimal level of granularity of recurrence. These form the basic building blocks of the architecture. In figure 5 and 6, coarser level of granularity modules based on level 2 and 3 modules replace more complex instruction sequences. A count of such modules at every level of granularity can be obtained for an entire algorithm. These statistics combined with the weights associated with each module based on the complexity of computation and data movements will help determine the extent of reconfiguration and the routing patterns on the processor. A library of hardware modules is developed for these patterns to facilitate the design of an embedded memory system, control logic and data paths of the reconfigurable processor. The tool also performs an analysis of the control structures. Control flow graphs include structures such as for loops, do while loops and if-then-else statements. Detecting conditional structures such as if-then-else statements exposes thread level parallelism.

Parallelism at the level of threads is a dominating factor in the design of hardware support for speculative execution. For example, the if-then-else control structure at the higher level of control such as:

```
if(GetVopQuantType(curr))
{
    BlockQuantMPEG(coeff_ind, QP, Mode, type, qmat, qcoeff_ind)
}
else
{
    BlockQuantH263(coeff_ind, QP, Mode, type, qcoeff_ind)
}
```

launches two threads **BlockQuantMPEG** and **BlockQuantH263** in parallel. At any point during the quantization process if the feedback to the encoder necessitates a change from one type to another, then an instantaneous switch can be applied due to the parallel running thread. In the assembly level we are able to detect the beginning and ending points of *if* and *else* parts. We are also able to make this detection for any control statement including nested *for* loops or nested *if* statements or combinations of the two.

A hierarchical based detection mechanism helps in restricting the search for cluster based patterns in data flow graphs in order to perform the search in real time. Currently we are designing algorithms to detect the patterns in real time. Based on the library built from the recurring pattern analysis and the feasibility of routing, the partitioner assigns modules of the input application to either hardware or software. The router places the hardware modules on the chip, in order to minimize the amount of data movement and maximize resource utilization between reconfigurations.

Temporal partitioning divides the design into mutually exclusive, limited size segments such that the logic required to implement a segment is less than or equal to the logic capacity of a configurable processor. Such segments can be scheduled for execution in proper order to ensure correct overall execution. Since run time reconfiguration imposes the constraint of changing resource count from cycle to cycle, there arises a need to perform a two layered scheduling before subjecting the hardware implementable modules to temporal partitioning. The first layer performs time constrained scheduling. The second layer performs a scheduling on the results of the first layer with either circular or learning/rule-based algorithms. The results of this process are then subjected to temporal partitioning. A detailed complexity analysis of a variety of multimedia algorithms followed by the above methodology will enable the development of a fully automated hardware-software

co-design tool. This will facilitate the design of dynamically reconfigurable multimedia processor.

#### 4 Results and Conclusion

We have executed the pattern recognizer tool for various algorithms in MPEG4. We present the results obtained for motion estimation, macro block motion estimation, quantization motion compensation, Discrete Wavelet Transform (DWT), Inverse DWT, shape coding and sprite coding. From figure 7 load, move, and shift operations occur quite frequently reaching to almost 40 % in some cases. Our tool searches for combinations of G modules that are close to each other in the parse tree to form larger clusters. Then it detects how frequently these clusters are occurring in the code. As shown in figure 7 at granularity level one, since "load" operation has the highest frequency, the tool first searches for relationship between that module and others. The results in figure 8 show that in DWT a "load" operation (module  $G^1_3$ ) is more likely to be clustered with itself or  $G^1_2$ . On the other hand in motion compensation module  $G^1_3$  is more likely to be clustered with  $G^1_1$  hence forming level two ( $G^2_x$ ) clusters. The availability of this information a priori in the execution of each of the modules in MPEG4 video processing allows us to determine the resources required and frequency of usage of library modules. This will facilitate the design of interconnection and routing architecture design. In the future we will detect all possible combinations of library modules to determine the optimal level of granularity for hardware software partitioning and routing architecture. Following that, based on the library built from the recurring pattern analysis and the feasibility of routing, the partitioner assigns modules of the input application to either hardware or software. The router places the hardware modules on the chip, in order to minimize the amount of data movement and maximize resource utilization between reconfigurations. The proposed methodology enables efficient partitioning of resources for complex and time critical multimedia applications.

#### 5 Summary

There is an increase in demand for reconfigurable multimedia processors in applications such as MPEG4 video application. We have recently presented a methodology for designing a reconfigurable media processor. In this paper we have detailed the design of the parser and recurring pattern analyzer and shown the results for MPEG4 video processing.

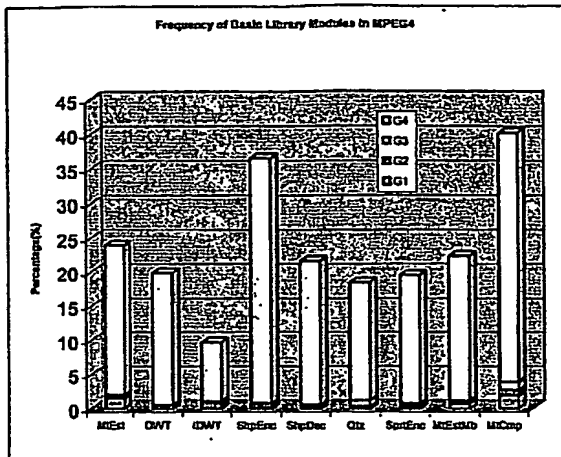


Figure 7

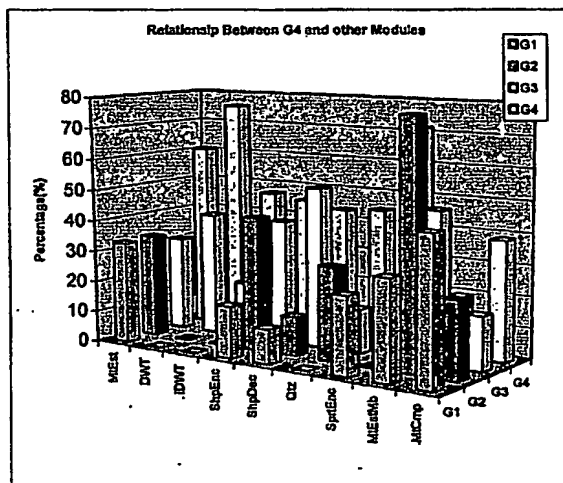
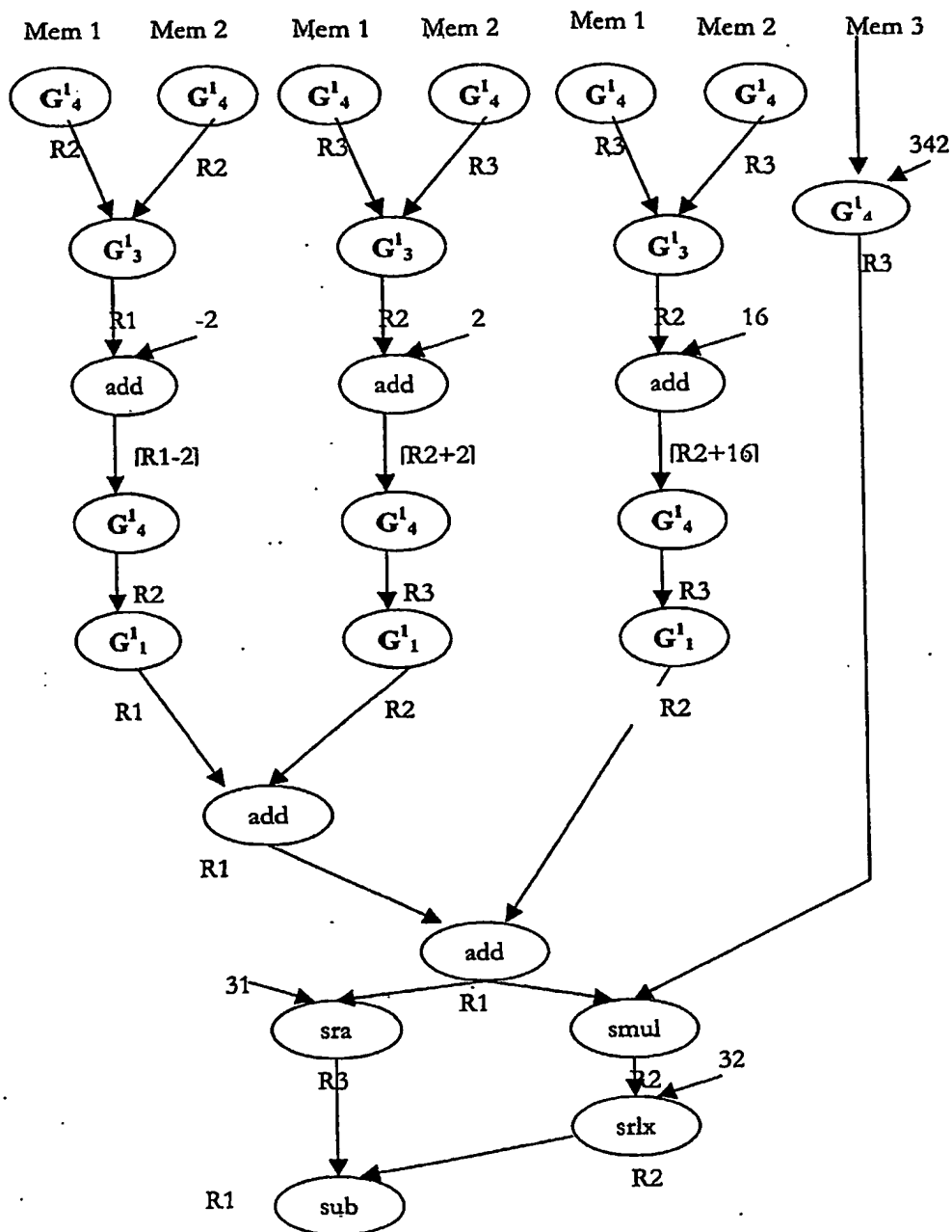


Figure 8

## References

1. "Reconfigurable Media Processing", Dasu A, Panchanathan S. Information Technology: Coding and Computing, 2001. Proceedings. International Conference on, 2001 Pages 300-304.
2. ISO/IEC JTC1/SC29/WG11, "MPEG-4 video verification model version 11.0," Doc. 2172, Mar. 1998.
3. "Complexity Analysis of MPEG-4 Video Profiles", A Master's thesis by C.N. Raghavendra. Arizona State University, 2000.
4. "Algorithms, Complexity Analysis and VLSI Architectures for MPEG 4 Motion Estimation", Peter Kuhn. Kluwer academic publishers.
5. "Reconfigurable Architectures for General-Purpose Computing", Andre DeHon. A.I Technical Report No. 1586. October 1996. Massachusetts Institute of Technology.
6. PipeRench: a coprocessor for streaming multimedia acceleration Goldstein, S.C.; Schmit, H.; Moe, M.; Budiu, M.; Cadambi, S.; Taylor, R.R.; Laufer, R. Computer Architecture, 1999. Proceedings of the 26th International Symposium on, 1999 Page(s): 28-39
7. MorphoSys: a reconfigurable architecture for multimedia applications Singh, H.; Ming-Hau Lee; Guangming Lu; Kurdahi, F.J.; Bagherzadeh, N.; Filho, E.M.C. Integrated Circuit Design, 1998. Proceedings. XI Brazilian Symposium on, 1998 Page(s): 134-139
8. CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit Ye, Z.A.; Moshovos, A.; Hauck, S.; Banerjee, P. Computer Architecture, 2000. Proceedings of the 27th International Symposium on, 2000 Page(s): 225-235
9. Integration of high-performance ASICs into reconfigurable systems providing additional multimedia functionality Blume, H.; Bluthgen, H.-M.; Henning, C.; Osterloh, P. Application-Specific Systems, Architectures, and Processors, 2000. Proceedings. IEEE International Conference on, 2000 Page(s): 66-75
10. An application-tailored dynamically reconfigurable hardware architecture for digital baseband processing Becker, J.; Pionteck, T.; Glesner, M. Integrated Circuits and Systems Design, 2000. Proceedings. 13th Symposium on, 2000 Page(s): 341-346
11. Coarse reconfigurable multimedia unit extension Wong, S.; Cotofana, S.; Vassiliadis, S. Parallel and Distributed Processing, 2001. Proceedings. Ninth Euromicro Workshop on, 2001 Page(s): 235-242
12. "Tooling Up for Reconfigurable System Design", Gordon Brebner. 1999 IEEE.
13. "Speeding up Program Execution Using Reconfigurable Hardware and a Hardware Function Library", Sitanshu Jain, M. Balakrishnan, Anshul Kumar, Shashi Kumar. 1997 IEEE.

Figure 3 Tree with level  $G^1_x$  modules

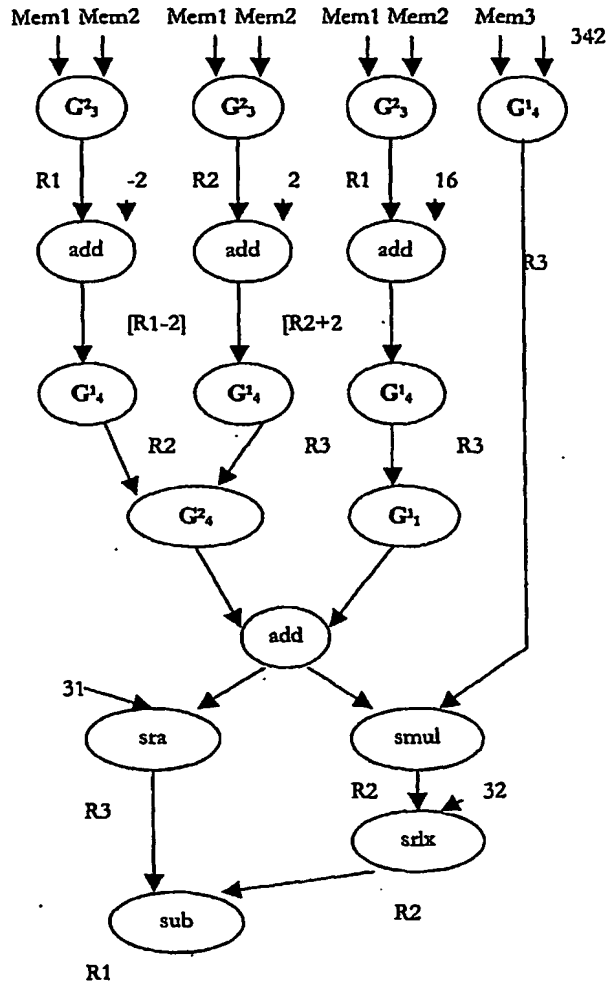


Figure 5

Tree with higher granularity level  
2 and 1 modules

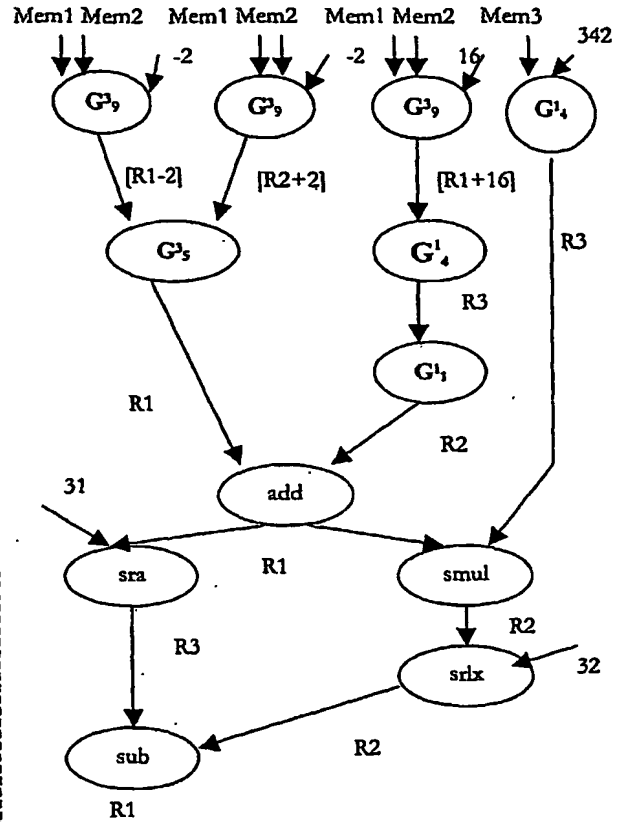


Figure 6

Tree with level 3 and 1 modules

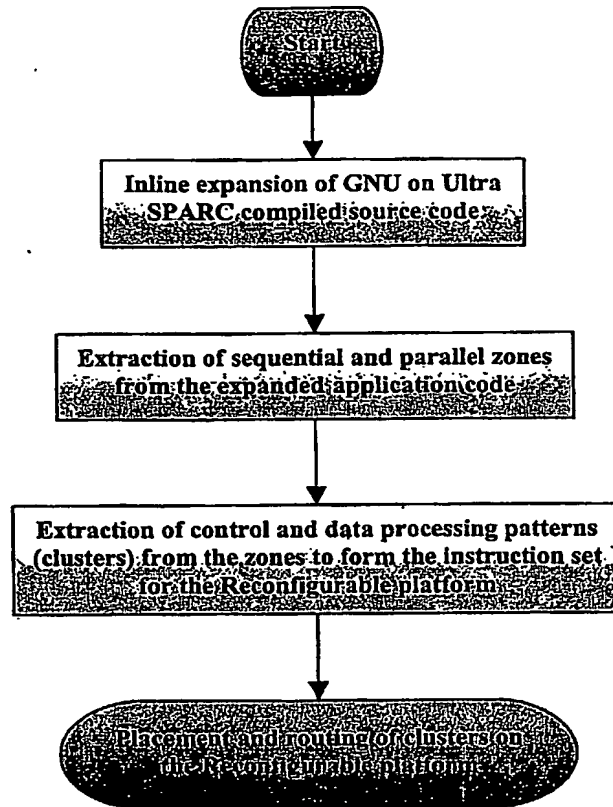






**Detailed description of invention**  
**Control and data processing pattern extraction from source code for multimedia applications**

The following flow chart gives the chronological sequence of the main tasks involved



High level C code of the target application is converted to assembly code as first step of the designing a reconfigurable processor for it. Then the assembly code is analyzed to detect the recurring patterns. However before that step the assembly code needs to be preprocessed. The following section explains the expander tool implemented for that preprocessing purpose.

First the need for preprocessing is explained with an example. The sample code includes main.c, findsum.c and findsub.c as shown in appendix-A. The assembly codes of those functions (main.s, findsum.s, findsub.s) are shown in appendixes B through D. In main.s we see that findsum() function is called twice. Trying to detect the recurring patterns in main.s might not give us all possible patterns since the assembly code of called functions are missing. For that purpose the main function needs to be expanded meaning that assembly code of all called functions need to be inserted to the place where it is called.

The steps included in expander() function is explained next:

Input : .s files ( main.s, findsum.s findsub.s )

#### Step-1 : Parse .s files

In this step for each .s file a doubly linked list is created where each node stores one instruction with operands and each node has pointers to the previous and next instructions in the assembly code. Parser ignores all commented out lines, lines without instructions except the labels such as

Main:

.LL3:

Each label starting with .LL is replaced with a unique number (unique over all functions )

#### Step-2: Expand

Each .s file has been parsed and stored in separate linked lists. In this step expander moves through the nodes of linked list that stores main.s. If a function call is detected that function is searched through all linked lists. When it is found, that function from beginning to the end is copied and inserted into the place where it is called. Then expander continues moving through the nodes where it stopped. Expanding continues until the end of main.s is reached. Note that if an inserted function is also calling some other function expander also expands it until every called function is inserted to the right place.

In the sample code, main() function is calling the findsum() function twice and findsum() function is calling the findsub() function. The output of the expander is shown in Appendix-E.

#### Step-3: Create control flow linked list

The main.s has been expanded and stored in a doubly linked list, next step is to create another doubly linked list ( control\_flow\_linked\_list ) that stores the control flow information. This will be used to analyze the control flow structure of the application

code, to detect the starting and ending places of functions, control structures ( loops, if..else statements, etc. ). The control flow linked list helps to position the place of the beginning and ending of each function, and control structures without having to search them through the expanded linked list.

This module moves through nodes of the expanded linked list and if a node belongs to a :

- label
- function
- conditional
- unconditional branch

a new node is created to be appended to the control flow linked list by setting the member pointers as defined below.

If the current node is a

- **function label**  
A pointer to the expanded list pointing to the function label node  
A pointer to the expanded list pointing to the beginning of the function (the next node of the function label node)  
A pointer to the expanded list pointing to the end of the function  
And node type is set to "function".
- **label**  
A pointer to the expanded list pointing to the function label node  
A pointer to the expanded list pointing to the beginning of the label (the next node of the label node )  
And node type is set to "square".
- **unconditional branch (b)**  
A pointer to the expanded list pointing to the branch node  
A pointer to the control flow linked list pointing to the node that stores the matching target label of the branch instruction.  
And node type is set to "dot"
- **conditional branch (bne, ble, bge, ...etc)**  
A pointer to the expanded list pointing to the branch node  
A pointer to the control flow linked list pointing to the node that stores the matching target label of the branch instruction.  
And node type is set to "circle".

The control flow linked list output is shown in Appendix-F for the findsum.s function.

## Appendix A

### C code

```
#include<stdio.h>

void main()
{
    int i,j,k,l;

    i = 10;
    j = 1* 4;

    if ( j > 5 )
    {
        k=findsum(i,j);
        l = 4+k;
    }
    else
    {
        k = findsum(i,j);
        l = k*10;
    }
}

int findsum(int a,int b)
{
    int i,j,k;

    k=4;
    for(i=0;i<10;i++)
        k = k + 1;

    j = findsub(k,a);
    return j;
}

int findsub(int x,int y)
{
    int t;

    t = x-y;

    return(t);
}
```

## Appendix B

Main.s

```
.file "main.c"
gcc2_compiled.:
.section ".text"
.align 4
.global main
.type main,#function
.proc 020
main:
    !#PROLOGUE# 0
    save %sp, -128, %sp
    !#PROLOGUE# 1
    mov    10, %o0
    st     %o0, [%fp-20]
    mov    4, %o0
    st     %o0, [%fp-24]
    ld     [%fp-24], %o0
    cmp    %o0, 5
    ble    .LL3
    nop
    ld     [%fp-20], %o0
    ld     [%fp-24], %o1
    call   findsum, 0
    nop
    st     %o0, [%fp-28]
    ld     [%fp-28], %o0
    add    %o0, 4, %o1
    st     %o1, [%fp-32]
    b      .LL4
    nop
.LL3:
    ld     [%fp-20], %o0
    ld     [%fp-24], %o1
    call   findsum, 0
    nop
    st     %o0, [%fp-28]
    ld     [%fp-28], %o0
    mov    %o0, %o2
    sll    %o2, 2, %o1
    add    %o1, %o0, %o1
    sll    %o1, 1, %o0
    st     %o0, [%fp-32]
.LL4:
.LL2:
    ret
    restore
.LLfel:
.size    main,.LLfel-main
.ident   "GCC: (GNU) 2.95.2 19991024 (release)"
```

## Appendix C

### Findsum.s

```

        .file "findsum.c"
gcc2_compiled.:
.section    ".text"
        .align 4
        .global findsum
        .type findsum,#function
        .proc 04
findsum:
        !#PROLOGUE# 0
        save %sp, -128, %sp
        !#PROLOGUE# 1
        st    %i0, [%fp+68]
        st    %i1, [%fp+72]
        mov   4, %o0
        st    %o0, [%fp-28]
        st    %g0, [%fp-20]
.LL3:
        ld    [%fp-20], %o0
        cmp   %o0, 9
        ble   .LL6
        nop
        b     .LL4
        nop
.LL6:
        ld    [%fp-28], %o0
        add   %o0, 1, %o1
        st    %o1, [%fp-28]
.LL5:
        ld    [%fp-20], %o0
        add   %o0, 1, %o1
        st    %o1, [%fp-20]
        b     .LL3
        nop
.LL4:
        ld    [%fp-28], %o0
        ld    [%fp+68], %o1
        call  findsub, 0
        nop
        st    %o0, [%fp-24]
        ld    [%fp-24], %o0
        mov   %o0, %i0
        b     .LL2
        nop
.LL2:
        ret
        restore
.LLfel:
        .size findsum, .LLfel-findsum
        .ident    "GCC: (GNU) 2.95.2 19991024 (release)"

```

## Appendix D

### Findsub.s

```
.file "findsub.c"
gcc2_compiled.:
.section ".text"
.align 4
.global findsub
.type findsub,#function
.proc 04
findsub:
!#PROLOGUE# 0
save %sp, -120, %sp
!#PROLOGUE# 1
st %i0, [%fp+68]
st %i1, [%fp+72]
ld [%fp+68], %o0
ld [%fp+72], %o1
sub %o0, %o1, %o0
st %o0, [%fp-20]
ld [%fp-20], %o0
mov %o0, %i0
b .LL2
nop
.LL2:
ret
restore
.LLfel:
.size findsub,.LLfel-findsub
.ident "GCC: (GNU) 2.95.2 19991024 (release)"
```

## Appendix E

### Expanded main function

Function main BEGINS here

```
save %sp -128 %sp
mov 10 %o0
st %o0 [%fp-20]
mov 4 %o0
st %o0 [%fp-24]
ld [%fp-24] %o0
cmp %o0 5
ble 0
nop
ld [%fp-20] %o0
ld [%fp-24] %o1
Function findsum BEGINS here
```

```
save %sp -128 %sp
st %i0 [%fp+68]
st %i1 [%fp+72]
mov 4 %o0
st %o0 [%fp-28]
st %g0 [%fp-20]
4
ld [%fp-20] %o0
cmp %o0 9
ble 5
nop
b 6
nop
5
ld [%fp-28] %o0
add %o0 1 %o1
st %o1 [%fp-28]
7
ld [%fp-20] %o0
add %o0 1 %o1
st %o1 [%fp-20]
b 4
nop
6
ld [%fp-28] %o0
ld [%fp+68] %o1
Function findsub BEGINS here
```

```
save %sp -120 %sp
st %i0 [%fp+68]
st %i1 [%fp+72]
ld [%fp+68] %o0
ld [%fp+72] %o1
sb %o0 %o1 %o0
st %o0 [%fp-20]
```

```
ld [%fp-20] %o0
mov %o0 %i0
b 10
nop
10
ret
restore
11
Function findsub ENDS here
findsub .LLfe1-findsub
nop
st %o0 [%fp-24]
ld [%fp-24] %o0
mov %o0 %i0
b 8
nop
8
ret
restore
9
Function findsum ENDS here
findsum .LLfe1-findsum
nop
st %o0 [%fp-28]
ld [%fp-28] %o0
add %o0 4 %o1
st %o1 [%fp-32]
b 1
nop
0
ld [%fp-20] %o0
ld [%fp-24] %o1
Function findsum BEGINS here

save %sp -128 %sp
st %i0 [%fp+68]
st %i1 [%fp+72]
mov 4 %o0
st %o0 [%fp-28]
st %o0 [%fp-20]
4
ld [%fp-20] %o0
cmp %o0 9
ble 5
nop
b 6
nop
5
ld [%fp-28] %o0
add %o0 1 %o1
st %o1 [%fp-28]
7
ld [%fp-20] %o0
add %o0 1 %o1
st %o1 [%fp-20]
b 4
nop
```

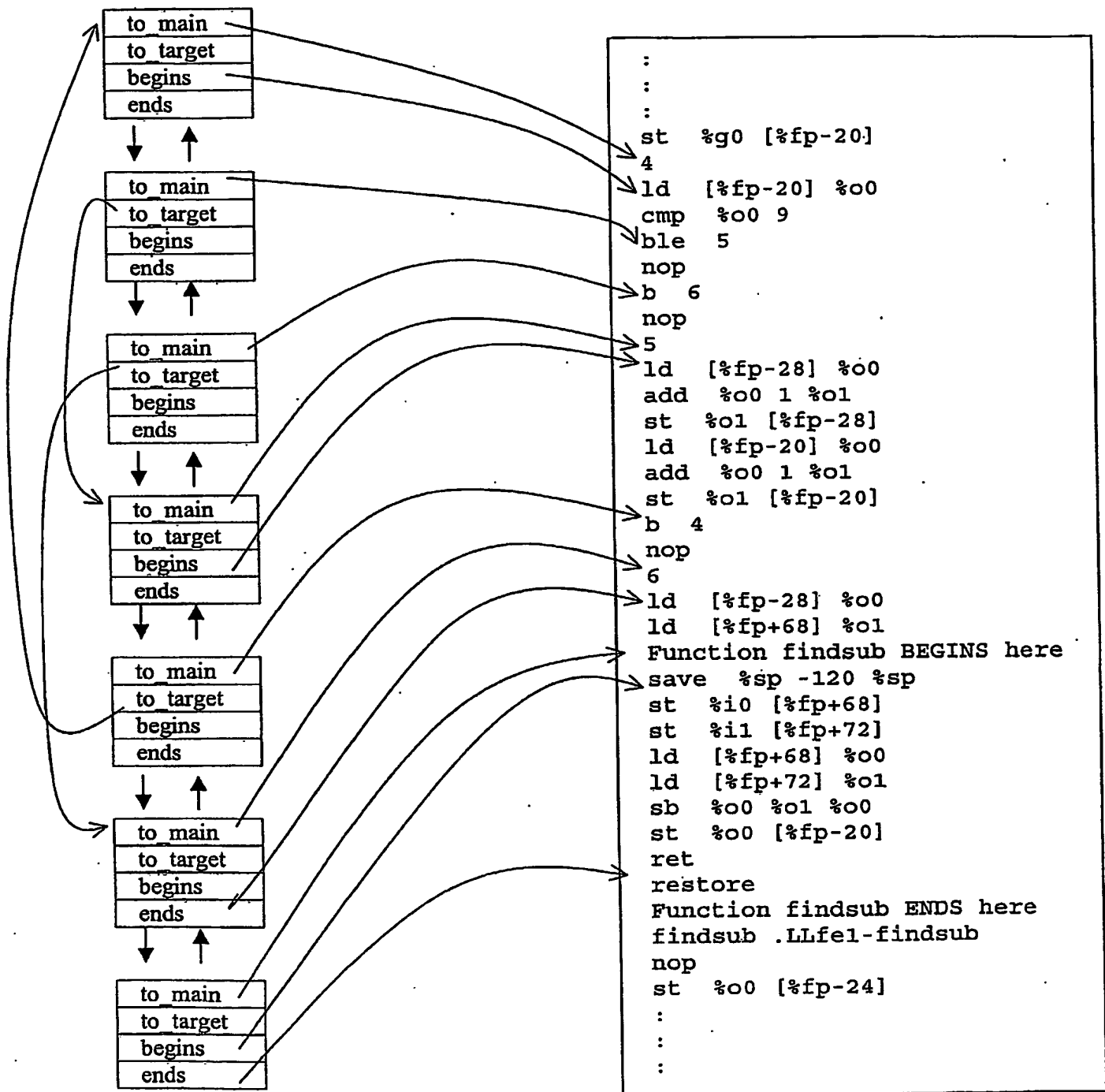
```
6
ld [%fp-28] %o0
ld [%fp+68] %o1
Function findsub BEGINS here

save %sp -120 %sp
st %i0 [%fp+68]
st %i1 [%fp+72]
ld [%fp+68] %o0
ld [%fp+72] %o1
sb %o0 %o1 %o0
st %o0 [%fp-20]
ld [%fp-20] %o0
mov %o0 %i0
b 10
nop
10
ret
restore
11
Function findsub ENDS here
findsub .LLfel-findsub
nop
st %o0 [%fp-24]
ld [%fp-24] %o0
mov %o0 %i0
b 8
nop
8
ret
restore
9
Function findsum ENDS here
findsum .LLfel-findsum
nop
st %o0 [%fp-28]
ld [%fp-28] %o0
mov %o0 %o2
sll %o2 2 %o1
add %o1 %o0 %o1
sll %o1 1 %o0
st %o0 [%fp-32]
1
2
ret
restore
3
Function main ENDS here
```

## Appendix F

## Control flow linked list

## Main linked list



The control structure linked list (which essentially represents the control flow graph of the candidate algorithm) is modified to create a tree based structure (called "control flow tree"). An algorithmic description of the conversion process is described through the function 'Treeise'.

Function name: Treeise.

The linear linked list provided by ali, consists of most of the desired links between the nodes. But certain changes need to be incorporated.

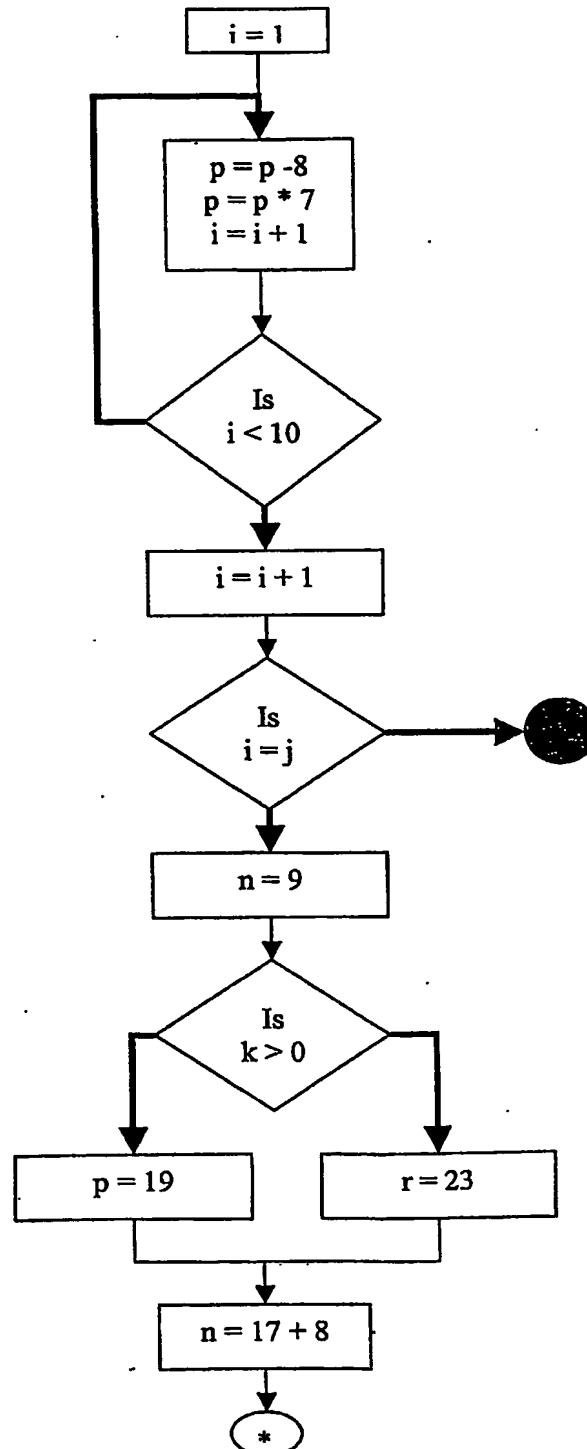
- The pointers from unconditional branch nodes (also called "dot" nodes) to the next node in the list need to be disconnected and made NULL. Hence for the "dot" node:  
   node→ next = NULL  
   for the following node:  
   node→ previous = NULL  
   {Exception: if the next node of the "dot" node is itself the target node !}
- The target nodes of the unconditional branches need to be marked as "Possible Exit" nodes. These "Exit" classes of nodes are a subset of the regular "Target" or "Square" nodes.
- If unconditional branch node's rank is higher than target node's rank (indicating a feed back or loop), disconnect the link and mark as NULL.  
   Hence for the "dot" node:  
   node→ to\_target = NULL  
   But before disconnecting, mark target→ next (which should be a circle) as "loop node".
- In a special case, if an unconditional branch and a square share the same node, then the target of that unconditional branch is declared as an exit square with a loop type (because, instructions following this square, comprise the meat of the do-while loop). This exit square, will not have its next→ pointing to a circle. The circle is accessed through the dot node using the previous→ pointer. Then it is marked off as type loop.
- If a "Possible Exit" node has 2 valid input pointers, and rank of both source pointers is lesser than the node in consideration, then it is an "Exit" node and, disconnect the link to the corresponding "dot" node, and hence also mark that "dot" node's target pointer to NULL. In other words, if the node→ previous pointer of the "square/target" node of the "dot" node does not point to the "dot" node, then it has 2 valid pointers.  
   Hence for the "dot" node:  
   node→ to\_target = NULL

For the high level code in the Figure 1 below (corresponding flow chart in Figure 2), the control flow linked list is as shown in Figure 3. After modifications to this linked list, function 'treeise' generates a tree structure indicated in figure 4.

```
#include<stdio.h>
void main()
{
    int i=0,j=0,k=0,l=0,m=0,n=0,p=0,r=0;

    for(i=1;i<10;i++)
    {
        p = p - 8;
        p = p * 7;
    }
    i = i + 1;
    if(i==j)
    {
        n = 9;
        if (k>0)
        {
            p = 19;
        }
        else
        {
            r = 23;
        }
        n = 17 + 8;
    }
    else
    {
        l = 10;
        m = n + r;
    }
    k = k - 14;
    k = 7 - 8 * p;
    while(i<p)
    {
        p = p * 20;
        p = p - 7;
        while(k == 8)
        {
            p = p + 17;
            i = i * p;
        }
        p = p - 23;
    }
    m = m + 5;
    n = n + 4;
}
```

Figure 1: Candidate algorithm



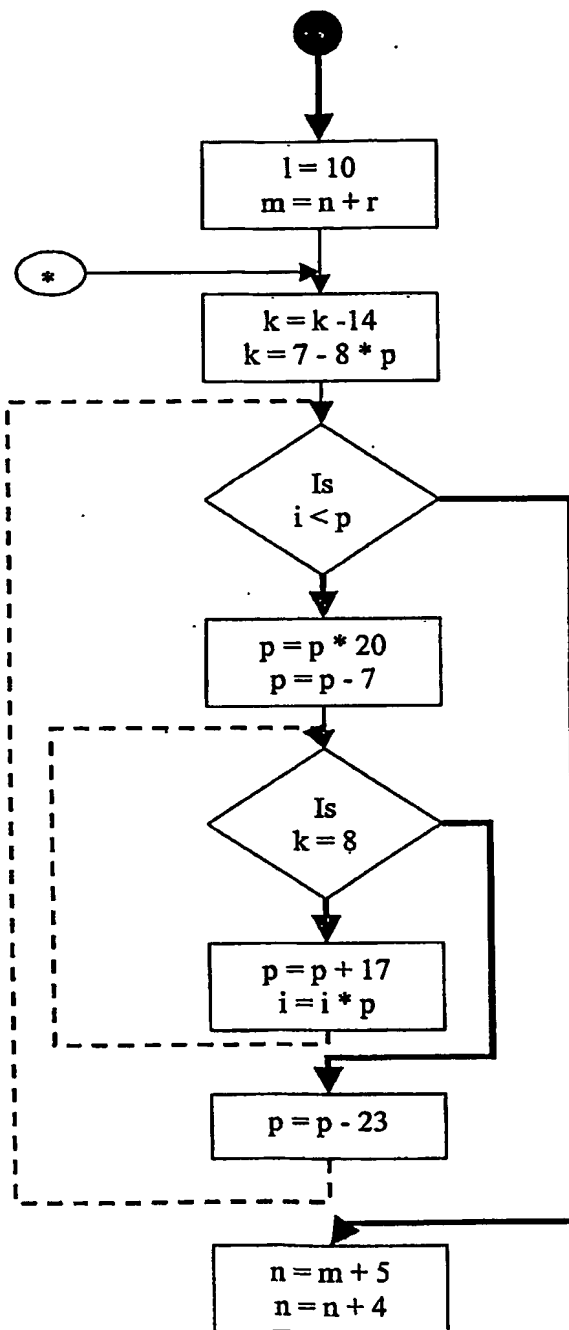


Figure 2: Flow chart of candidate algorithm

The gcc (version 2.95.2) compiled code for the UltraSPARC architecture is as follows:

```

        .file    "loop_pattern4.c"
gcc2_compiled.:
        .global  .umul
.section     ".text"
        .align  4
        .global  main
        .type    main,#function
        .proc    020
main:
        !#PROLOGUE# 0
        save    %sp, -144, %sp
        !#PROLOGUE# 1
        st      %g0, [%fp-20]           ground
        st      %g0, [%fp-24]
        st      %g0, [%fp-28]
        st      %g0, [%fp-32]
        st      %g0, [%fp-36]
        st      %g0, [%fp-40]
        st      %g0, [%fp-44]
        st      %g0, [%fp-48]
        mov     1, %o0
        st      %o0, [%fp-20]
.LL3:
        ld      [%fp-20], %o0           square 3
        cmp     %o0, 9
        ble     .LL6                   circle 6
        nop
        b       .LL4                   dot 4
        nop
.LL6:
        ld      [%fp-44], %o0           square 6
        add     %o0, -8, %o1
        st      %o1, [%fp-44]
        ld      [%fp-44], %o0
        mov     %o0, %o1
        sll     %o1, 3, %o2
        sub     %o2, %o0, %o0
        st      %o0, [%fp-44]
.LL5:
        ld      [%fp-20], %o0           square 5
        add     %o0, 1, %o1
        st      %o1, [%fp-20]
        b       .LL3                   dot 3
        nop

```

```

.LL4:
    ld    [%fp-20], %o0      square 4
    add   %o0, 1, %o1
    st    %o1, [%fp-20]
    ld    [%fp-20], %o0
    ld    [%fp-24], %o1
    cmp   %o0, %o1
    bne   .LL7              circle 7
    nop
    mov   9, %o0
    st    %o0, [%fp-40]
    ld    [%fp-28], %o0
    cmp   %o0, 0
    ble   .LL8              circle 8
    nop
    mov   19, %o0
    st    %o0, [%fp-44]
    b     .LL9              dot 9
    nop

.LL8:
    mov   23, %o0      square 8
    st    %o0, [%fp-48]

.LL9:
    mov   25, %o0      square 9
    st    %o0, [%fp-40]
    b     .LL10         dot 10
    nop

.LL7:
    mov   10, %o0      square 7
    st    %o0, [%fp-32]
    ld    [%fp-40], %o0
    ld    [%fp-48], %o1
    add   %o0, %o1, %o0
    st    %o0, [%fp-36]

.LL10:
    ld    [%fp-28], %o0      square 10
    add   %o0, -14, %o1
    st    %o1, [%fp-28]
    ld    [%fp-44], %o0
    mov   %o0, %o1
    sll   %o1, 3, %o0
    mov   7, %o1
    sub   %o1, %o0, %o0
    st    %o0, [%fp-28]

.LL11:
    ld    [%fp-20], %o0      square 11

```

ld	[%fp-44], %o1	
cmp	%o0, %o1	
bl	.LL13	circle 13
nop		
b	.LL12	dot 12
nop		
.LL13:		
ld	[%fp-44], %o0	square 13
mov	%o0, %o2	
sll	%o2, 2, %o1	
add	%o1, %o0, %o1	
sll	%o1, 2, %o0	
st	%o0, [%fp-44]	
ld	[%fp-44], %o0	
add	%o0, -7, %o1	
st	%o1, [%fp-44]	
.LL14:		
ld	[%fp-28], %o0	square 14
cmp	%o0, 8	
be	.LL16	circle 16
nop		
b	.LL15	dot 15
nop		
.LL16:		
ld	[%fp-44], %o0	square 16
add	%o0, 17, %o1	
st	%o1, [%fp-44]	
ld	[%fp-20], %o0	
ld	[%fp-44], %o1	
call	.umul, 0	
nop		
st	%o0, [%fp-20]	
b	.LL14	dot 14
nop		
.LL15:		
ld	[%fp-44], %o0	square 15
add	%o0, -23, %o1	
st	%o1, [%fp-44]	
b	.LL11	dot 11
nop		
.LL12:		
ld	[%fp-36], %o0	square 12
add	%o0, 5, %o1	
st	%o1, [%fp-36]	
ld	[%fp-40], %o0	
add	%o0, 4, %o1	

```
    st    %o1, [%fp-40]
.LL2:
    ret
    restore
.LLfel:
    .size  main,.LLfel-main
    .ident "GCC: (GNU) 2.95.2 19991024 (release)"
```

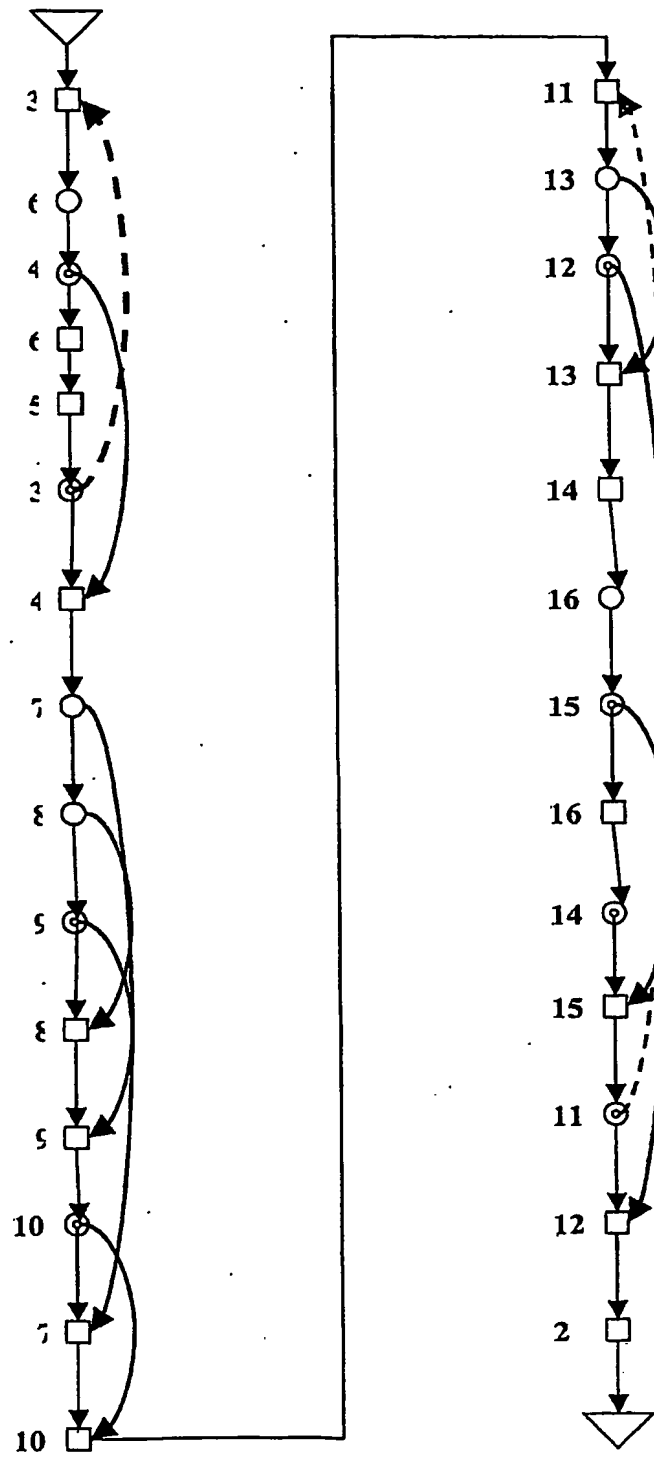


Figure 3: Control flow linked list

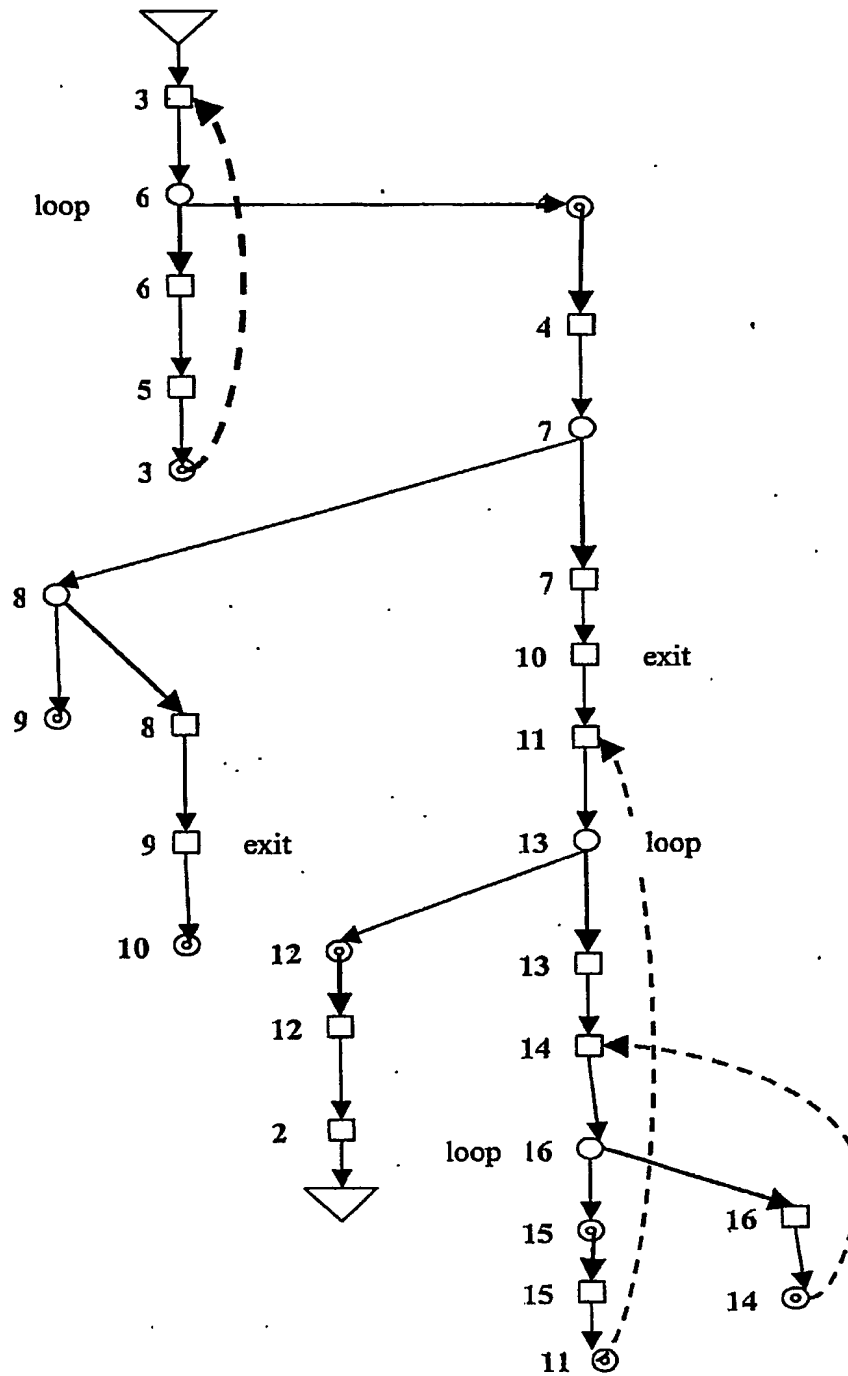


Figure 4: Tree structure obtained from the control flow linked list  
To meaningfully exploit and extract all possibilities of parallelism, pipelining and reconfiguration as stated in document dasu\_4, zones are identified in the tree structure. An algorithmic description of this process is described through the function 'Zonise' and is described below.

Function name: Zonise

A zone is any section of the compiled assembly code without any branch instructions. Zones are identified to meaningfully extract all possibilities of parallelism, pipelining and reconfiguration as stated in document dasu\_4. But to identify such sections, delimiters are needed. A delimiter can be any of the following types of nodes:

- (i) Circle
- (ii) Dot
- (iii) Exit square
- (iv) Square
- (v) Power
- (vi) Ground.

A 'Circle' can indicate the start of a new zone or the end of a zone. A 'Dot' can only indicate the end of a zone or a break in a zone. An 'Exit square' can indicate the start of a new zone or the end of a zone. A 'Square' can only indicate the continuation of a break in the current zone. A 'Power' can only indicate the beginning of the first zone. A 'Ground' can only indicate the end of a zone.

Figure 5 shows example zones to illustrate the use of delimiters, while Figure 6 shows approximately their correspondence with the high level code of the candidate algorithm.

Three zones, 1, 2, and 3 all share a common node, 'Circle 6'. This node is the end of Zone 1 and the start of zones 2 and 3. Zone 1 has the 'Power' node as its start, while Zone 6 has 'Ground' node as its end. The 'Dot 3' in Zone 3 indicates the end of that zone while 'Dot 4' indicates a break in Zone 2. This break is continued by 'Square 4'. In Zone 4, 'Square 9' indicates the end of the zone while it marks the start of Zone 5.

This function identifies zones in the tree, which is analogous to the numbering system in the chapter page of a book. Zones can have sibling zones (to identify if/else conditions, where in only one of the two possible paths can be taken {Zones 4 and 7 in Figure 1}) or child zones (to identify nested control structures {Zone 10 being child of zone 8 in Figure 1}). Zone types can be either simple or loopy in nature (to identify iterative loop structures). The tree is scanned node by node and decisions are taken to start a new zone or end an existing zone at key points such as circles, dots and exit squares. By default, when a circle is visited for the first time, the branch taken path is followed. But this node along with the newly started zone is stored in a queue for a later visit along the branch not taken path. When the tree has been traversed along the "branch taken" paths, the nodes with associated zones are popped out from the stack and traversed along their "branch not taken" paths. This is done till all nodes have been scanned and stack is empty.

Relationships:

A zone inside a control structure is the 'later child' of the zone outside the structure. Hence the zone outside a control structure and occurring before (in code sequence) the zone inside a control structure is a 'former parent' of the zone present inside.

But, the zone outside a control structure and occurring after (in code sequence) the zone inside the structure is referred to as the 'later parent'.

Similarly the child in this case would be a 'former child'.

A zone occurring after another zone and not related through a control structure is the 'next' of the earlier one.

Refer to the pseudo code for detailed description.

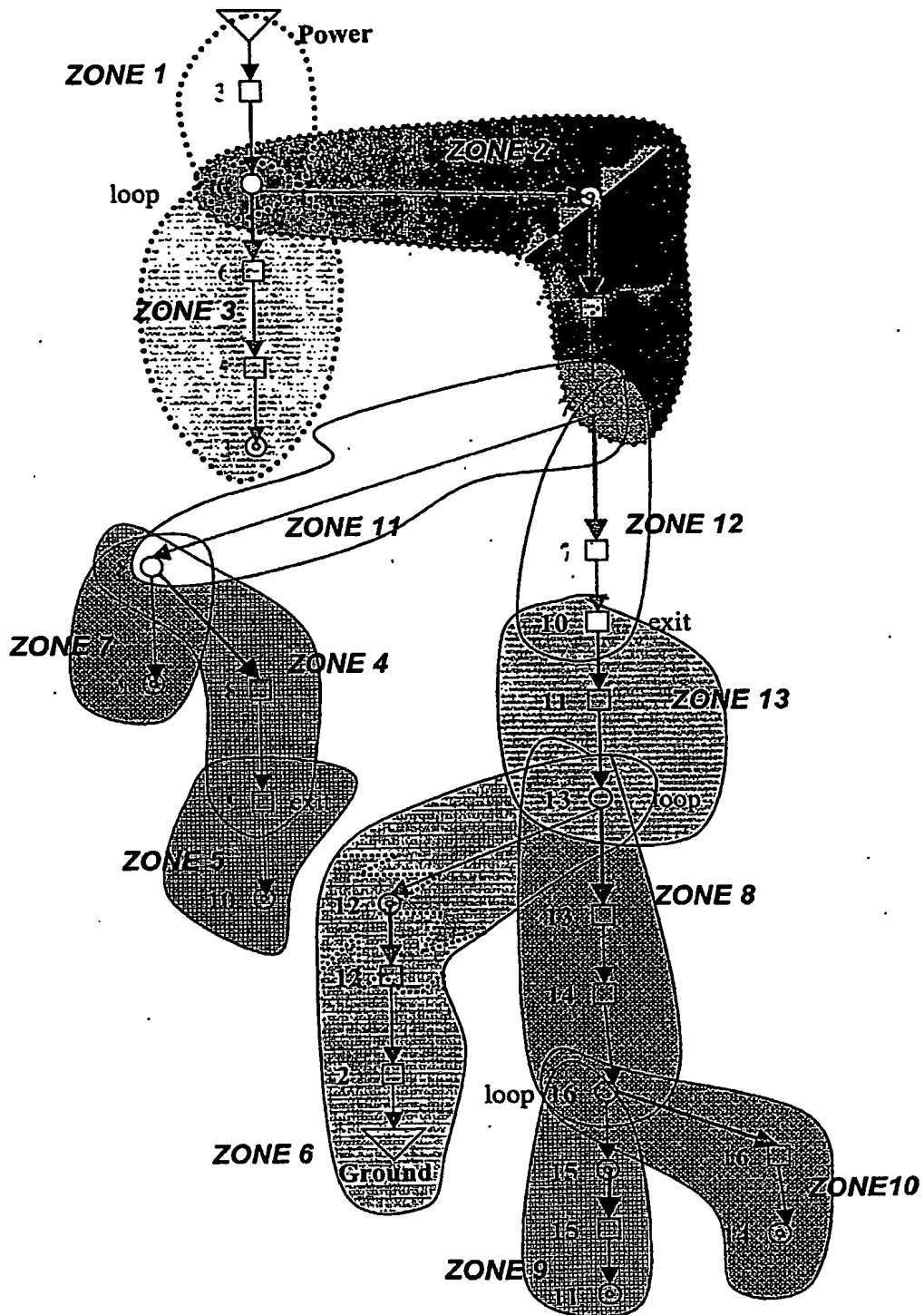


Figure 5: Zones in the tree structure

```

#include<stdio.h>
void main()
{
  int
  i=0,j=0,k=0,l=0,m=0,n=0,p=0,r=0;  ZONE 1

  for(i=1;i<10;i++)
  {
    p=p-8;  ZONE 3
    p=p-7;
    ZONE 2
    if(i==j)
    {
      n=9;  ZONE 11
      if (k>0)
      {
        p=19;  ZONE 7
      }
      else
      {
        ZONE 4
      }
      h=17-8;  ZONE 5
    }
    else
    {
      l=10;  ZONE 12
      m=n+1;
      ZONE 13
      k=2+14;
      k=7-8*p;
      while(i<p)
      {
        p=p*20;  ZONE 8
        p=p-7;
        while(k==8)
        {
          p=p+17;  ZONE 10
          i=p;
        }
        p=p-23;  ZONE 9
      }
      m=m+5;  ZONE 6
      n=n+4;
    }
  }
}

```

Figure 6: Approximate zone locations in the high level code

Pseudo code for 'zonise' Function:

Global variables: pop\_flag = 0, tree\_empty = 0;

Zonise (node) /\* input into the function is the current node, a starting node \*/

```
{
    while (tree_empty == 0) /* this loop goes on node by node in the tree till all node
        have been scanned */
    {
        if (node → type = circle)
        {
            if (pop_flag != set) /* pop flag is set when a pop operation is done */
            {
                /* an entry here means that the circle was encountered for the first
                time */
                /* so set the node → visited flag */
                /* close the zone */
                /* since u r entering a virgin circle, u cant create the new zone as a
                sibling to the one u just closed */
                /* if the zone u just closed, has a valid Anchor Point and if its of
                type Loop and if its visited flag is set, then u cannot create a
                child zone */
                /* accordingly create a new zone */
                /* set child as current zone */
                /* push this zone and the node into the queue */
                /* take the taken path for the node, i.e node = node → taken */
            }
            if (pop_flag = set)
            {
                /* an entry here means, that we r visiting a node and its associated
                zone, that have just been popped out form the queue, hence
                revisiting an old node */
                /* since this node has its visited flag as set, change that flag value
                to -1, so as to avoid any erroneous visit in the future */

                /* if node is of type Non Loop, then spawn a new sibling zone */
                /* if node is of type Loop, then spawn new zone as laterparent zone
                and mark zone type as loop */
                /* choose the not taken path for the node */
            }
        }
    }

    else if (node → type = exit square)
    {
        /* close the zone */
        /* if the closed zone has a parent, i.e zone → parent pointer is not NULL,
        then create a new zone with link to the parent zone as type next zone */
        /* if the closed zone does not have a parent, then spawn a new zone that is
```

```
        next to the closed zone */
        /* choose the not taken path for the node */
    }

    else if (node-> type is dot and node-> taken = NULL)
    {
        /* close zone */
        /* choose node to be considered next by popping out from the queue */
        /* in case the queue is empty, all nodes in tree have been scanned */
        /* set pop flag */
    }

    else if (node-> type = dot and node-> taken != NULL)
    {
        /* this is just a break in the current zone */
        /* create temp stop1 and tempstart1 pointers */
        /* choose node-> taken path */
    }
} /* end of the first while loop */
}
```

True code for 'zonise' Function:

Global variables: pop flag = 0, tree\_empty = 0;

Zonise (node) /\* input into the function is the current node, a starting node \*/

```
{
    while (tree_empty == 0)
    {
        if (node → type = circle)
        {
            if (pop_flag != set) /* pop flag is set when a pop operation is done */
            {
                /* an entry here means that the circle was encountered for the first
                time */
                /* so set the node → visited flag */
                node → visited = 1;
                /* close the zone */
                zone → stop = node;
                /* since u r entering a virgin circle, u cant create the new zone as a
                sibling to the one u just closed */
                /* if the zone u just closed, has a valid Anchor Point and if its of
                type Loop and if its visited flag is set, then u cannot create a
                child zone */
                /* accordingly create a new zone */
                zone → laterchild = create_new_zone();
                zone → laterchild → formerparent = zone;
                zone = zone → laterchild; /* setting child as current zone */
                /* push this zone and the node into the queue */
                push(zone, node);
                /* take the taken path for the node, i.e node = node → taken */
                node = node → taken;
            }
            if (pop_flag = set)
            {
                /* an entry here means, that we r visiting a node and its associated
                zone, that have just been popped out form the queue, hence
                revisiting an old node */

                /* since this node has its visited flag as set, change that flag value
                to -1, so as to avoid any erroneous visit in the future */
                node → visited_flag = -1;

                /* if node is of type Non Loop, then spawn a new sibling zone */
                if (node → type == NonLoop)
                {
                    zone → sibling = create_new_zone();
                    zone → sibling → previous_sibling = zone;
                    zone = zone → sibling;
                }
            }
        }
    }
}
```

```

    }
    /* if node is of type Loop, then spawn new zone as next zone */
    if (node->type == Loop)
    {
        zone->laterparent = create_new_zone();
        zone->laterparent->formerchild = zone;
        zone = zone->laterparent; /* setting newly created zone as
                                   current zone*/

        zone->type = loop;
    }
    /* choose the not taken path for the node */
    node = node->not_taken;
}

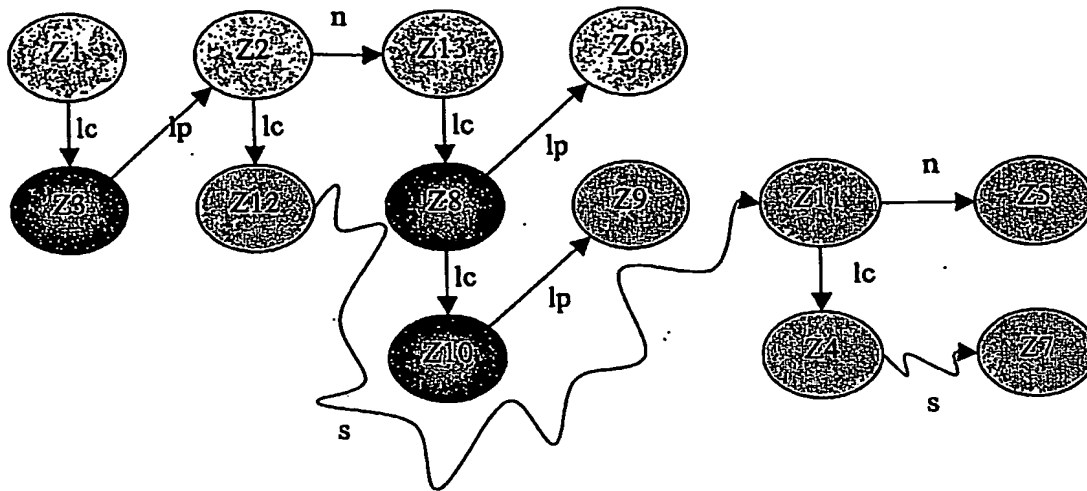
else if (node->type == exit square)
{
    /* close the zone */
    zone->stop = node;
    /* if the closed zone has a parent, i.e zone->parent pointer is not NULL,
       then create a new zone with link to the parent zone as type next zone */
    if (zone->parent != NULL)
    {
        zone->formerparent->next = create_new_zone();
        zone->formerparent->next->previous = zone->formerparent;
        zone = zone->formerparent->next;
    }
    /* if the closed zone does not have a parent, then spawn a new zone that is
       next to the closed zone */
    else
    {
        zone->next = create_new_zone();
        zone->next->previous = zone;
        zone = zone->next; /* setting newly created zone as current
                           zone*/
    }
    /* choose the not taken path for the node */
    node = node->not_taken;
}

else if (node->type is dot and node->taken == NULL)
{
    /* close zone */
    zone->stop = node;
    /* choose node to be considered next by popping out from the queue */
    /* in case the queue is empty, all nodes in tree have been scanned */

```

```
if (pop()→ queue_empty != 1)
{
    node = pop()→ the_node;
    zone = pop()→ the_zone;
    pop_flag = 1;
}
else
{
    printf("queue empty, all paths considered \n");
    tree_empty = 1;
}
}
else if (node→ type = dot and node→ taken != NULL)
{
    /* this is just a break in the current zone */
    zone→ temp_stop1 = node;
    zone→ temp_start1 = node→ taken;
    node = node→ taken;
}
}/* end of the first while loop */
}
```

After parsing through the tree structure thru function 'zonise' the zonal relationship as shown in Figure 7 is obtained.



S: sibling relationship  
 LC: later child relationship  
 Lp: later parent relationship  
 In all types, destination zone is (lc/s/lp) of source zone  
 The shaded zones are Loop types.

Figure 7: Initial Zone structure obtained after using function 'zonise'

This is referred to as the 'initial zone structure'. The term initial, is used because, some links need to be created and some existing ones, need to be removed. This process is explained in the section below.

#### Modification of the 'initial zone structure':

In Figure7, we see that Z1 can be connect to Z2 thru 'n'  
 Z12 can be connected to Z13 thru 'lp'  
 Z13 can be connected to Z6 thru 'n'  
 Z8 can be connected to Z9 thru 'n'  
 Z4 can be connected to Z5 thru 'lp'  
 Z5 can be connected to Z13 thru 'lp'  
 Z7 can be connected to Z5 thru 'lp'

But Z8's relationship to Z6 thru 'lp' is false, coz no node can have both 'n' and 'lp' links.  
 In such a case, remove the 'lp' link.

Rules to establish 'n' type links, if it doesn't exist:

If a zone (1) has an 'lc' link to zone (2), and if that zone (2) has a 'lp' link to a zone (3), then an 'n' link can be established between 1 and 3. This means that if zone (1) is of type 'loop', then zone (3) will now be classified as type 'loop' also.

Rules to establish 'lp' type links if it doesn't exist:

If a zone (1) has an 'fp' link to zone (2), and if that zone (2) has an 'n' link to a zone (3), then an 'lp' link can be established between 1 and 3

If a zone (1) has an 'lp' link to zone (2), and also has an 'n' link to zone (3), then first, remove the 'lp' link 'to zone (2)' from zone (1) and then, place an 'lp' link from zone (3) to zone (2).

This provides the 'comprehensive zone structure' as shown in Figure 8 (with cancelled links) and in Figure 9 (with all cancelled links removed).

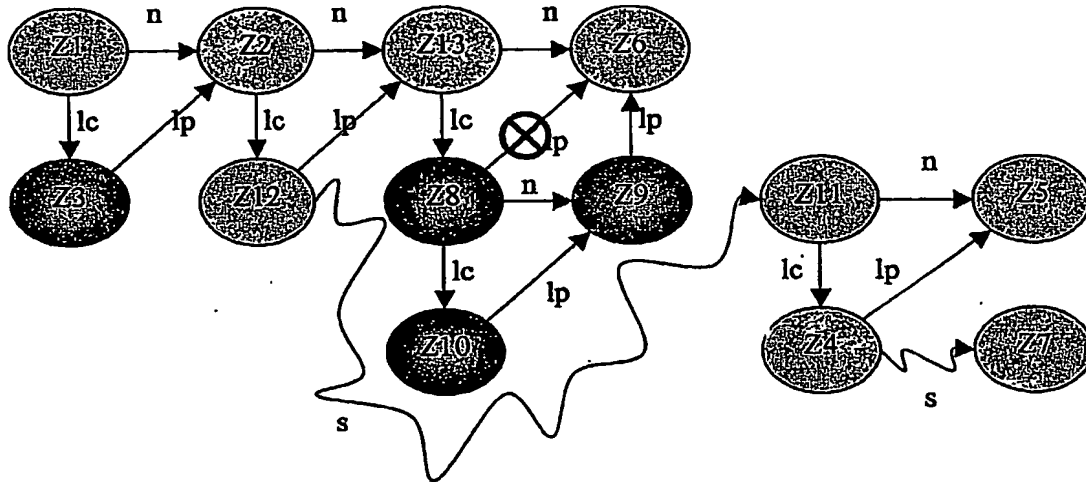


Figure 8: Comprehensive zone structure with cancelled links shown

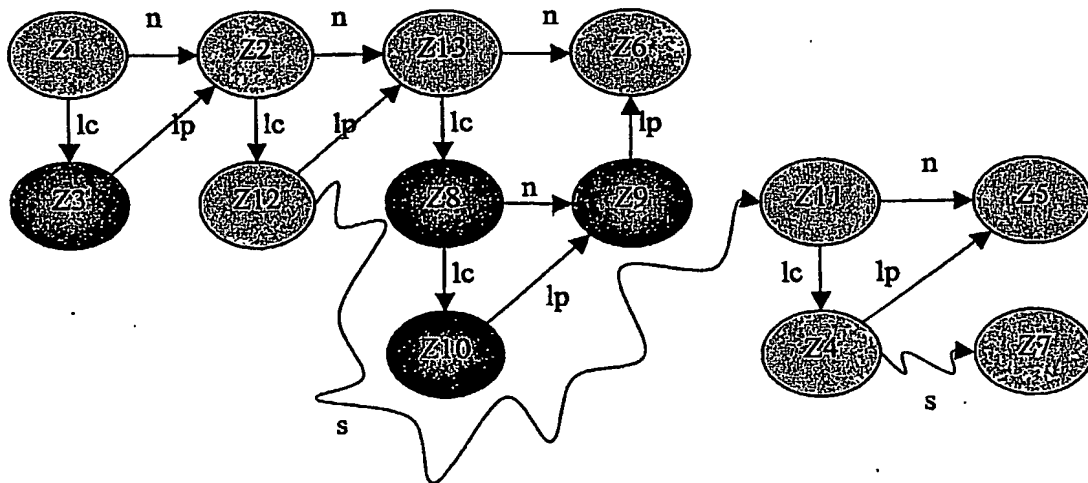


Figure 9: Comprehensive zone structure with cancelled links removed

To identify parallelism and hence compulsorily sequential paths of execution, the following approach is adopted.

Firstly, the comprehensive zone structure obtained, is ordered sequentially by starting at the first zone and traversing along an 'lc - lp' path. If a Sibling link is encountered it is given a parallel path. The resulting structure is shown in Figure 10.

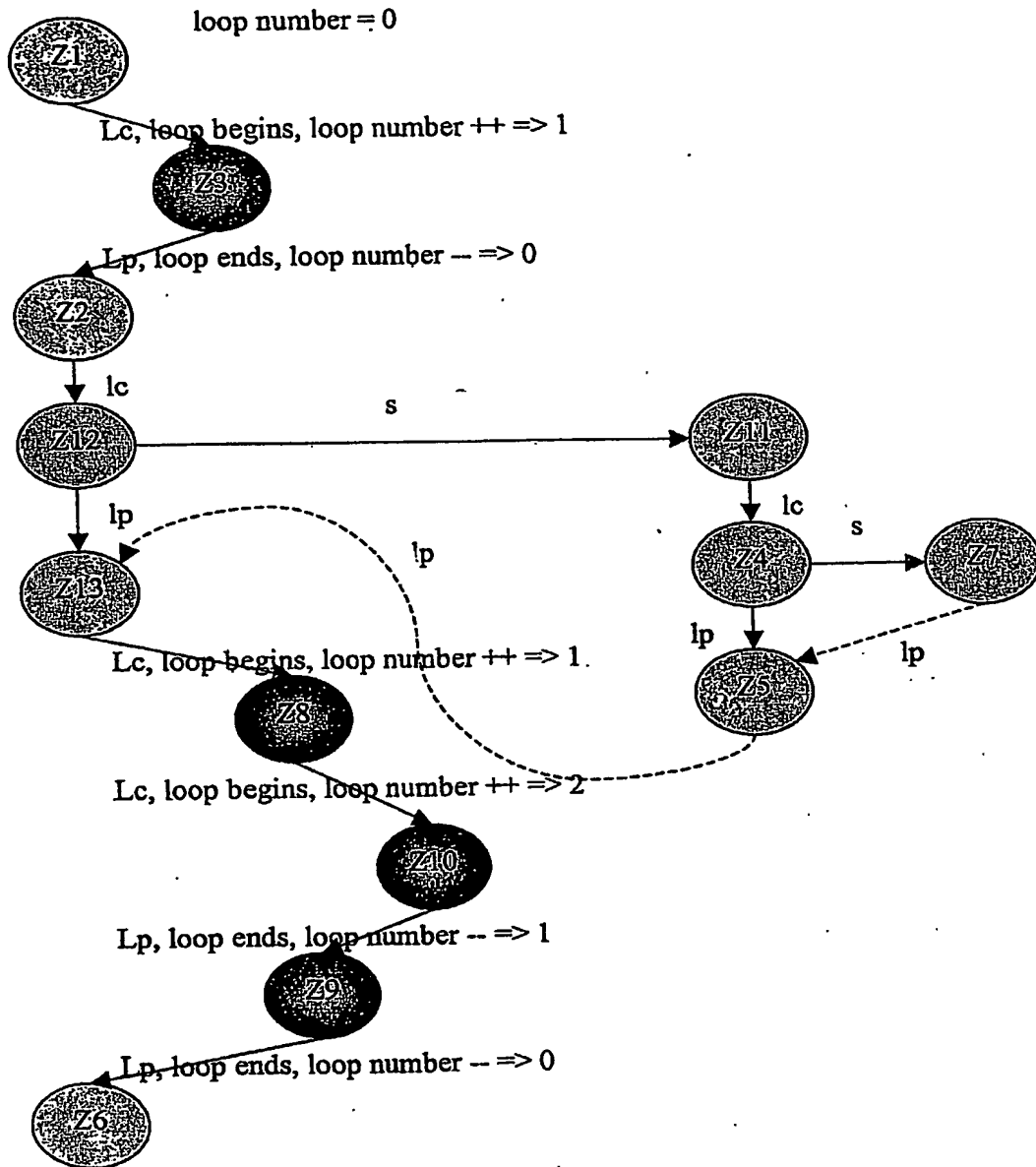


Figure 10: Sequentially ordered zones

To establish parallelism between a zone (1) of loop count A and its upper zone (2) of loop count B, where  $A < B$ , check for data dependency between zone 1 and all zones above it upto and including the zone with the same loop count as zone 2.

In the example above, to establish parallelism b/w zone 6 and zone 9, check for dependencies b/w zone 6 and 9, 10, 8. If there is no dependency then zone 6 is parallel to zone 8.

To establish parallelism between a zone (1) of loop count A and its upper zone (2) of loop count B, where  $A = B$ , direct dependency check needs to be performed.

To establish parallelism between a zone (1) of loop count A and its upper zone (2) of loop count B, where  $A > B$ , direct dependency check needs to be performed. Then, the zone (1) will now have to have an iteration count of (its own iteration count \* zone (2)'s iteration count).

When a zone rises like a bubble and is parallel with another zone in the primary path, and reaches a dependency, it is placed in a secondary path. No bubble in the secondary path is subjected to dependency testing.

After a bubble has reached its highest potential, and stays put in a place in the secondary path, the lowest bubble in the primary path is checked for dependency on its upper fellow.

If the upper bubble happens to have a different loop count number, then as described earlier, testing is carried out. In case a parallelism cannot be obtained, then this bubble, is clubbed with the set of bubbles ranging from its upper fellow, till and inclusive of the bubble up the chain with the same loop count as its upper fellow.

A global i/o parameter set is created for this new coalition.

Now this coalition will attempt to find dependencies with its upper fellow.

The loop count for this coalition will be bounding zone's loop count.

Any increase in the iteration count of this coalition will reflect on all zones inside it.

In case a bubble wants to rise above another one which has a sibling/ reverse sibling link, there will be speculative parallelism.

The algorithm should start at multiple points, one by one.

These points can be obtained by starting from the top zone and traversing down, till a sibling split is reached. Then this zone should be remembered, and one of the paths taken.

This procedure is similar to the stack saving scheme used earlier in the zonise function.

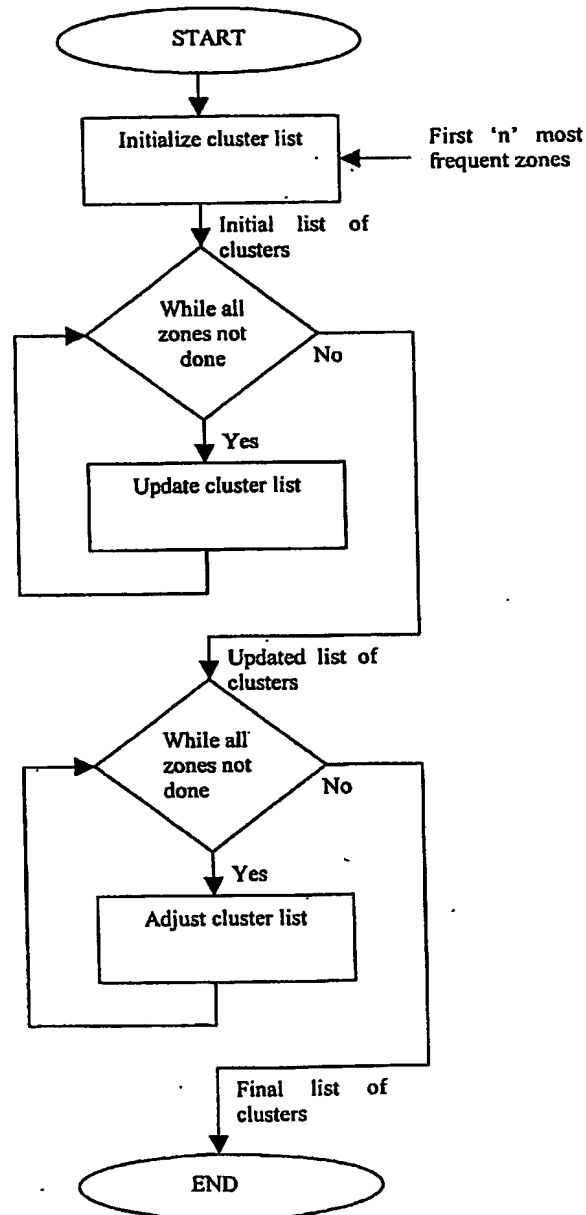
Low Average Latency

3-4 Cycles for Integer/FP/VIS Add, Subtract, Multiply, and Logic

17-29 Cycles for FP Divide and Square Root

### Function Clusterise

Each of the zones that are generated by the "zonise" function described above consists of a sequential set of instructions. The aim of this function "clusterise" is to detect the list of clusters that are most suitable to be implemented in hardware. Data dependency between the clusters will also be provided. Such a list of clusters will be different at different points of time as reconfiguration between clusters and the data path will give rise to a new list.



## Definitions:

1. **Cluster:** A cluster is defined to be a group of instruction sequences. The length of the cluster can vary between 5 and 10. Let L be the length of the cluster. Given below is the list of all possible instruction clusters that can be implemented using a single hardware module, for a given value of L. Enable bits will be provided with the hardware implementation of each instruction. This can be used to include that instruction in the cluster whenever it is required. This reduces the number of reconfigurations to be done.

**L = 5; e.g. 1,2,3,4,5**

The group of instruction sequences is as follows:

1,2,3,4,5  
1,2,3,4; 1,2,3,5; 1,2,4,5; 1,3,4,5; 2,3,4,5;  
1,2,3; 1,2,4; 1,3,4; 2,3,4; etc

**L = 6; e.g. 1,2,3,4,5,6**

The group of instruction sequences is as follows:

1,2,3,4,5,6  
1,2,3,4,5; 1,2,3,4,6; etc  
1,2,3,4; 1,2,3,5; 1,2,5,6; etc

**L = 7; e.g. 1,2,3,4,5,6,7**

The group of instruction sequences is as follows:

1,2,3,4,5,6,7  
1,2,3,4,5,6; 1,2,3,4,6,7; etc  
1,2,3,4,5; 1,2,3,5,6; 1,2,5,6,7; etc

**L = 8; e.g. 1,2,3,4,5,6,7,8**

The group of instruction sequences is as follows:

1,2,3,4,5,6,7,8  
1,2,3,4,5,6,7; 1,2,3,4,6,7,8; etc  
1,2,3,4,5,8; 1,2,3,5,6,8; 1,2,5,6,7,8; etc  
1,2,3,4,5; 1,2,3,4,6; etc

**L = 9; e.g. 1,2,3,4,5,6,7,8,9**

The group of instruction sequences is as follows:

1,2,3,4,5,6,7,8,9  
1,2,3,4,5,6,7,8; 1,2,3,4,5,6,7,9; etc  
1,2,3,4,5,6,7; 1,2,3,4,6,7,8; etc  
1,2,3,4,5,8; 1,2,3,5,6,8; 1,2,5,6,7,8; etc

**L = 10; e.g. 1,2,3,4,5,6,7,8,9,10**

The group of instruction sequences is as follows:

1,2,3,4,5,6,7,8,9,10  
1,2,3,4,5,6,7,8,9; 1,2,3,4,5,6,7,8,10; etc

1,2,3,4,5,6,7,8; 1,2,3,4,6,7,8,9; etc  
 1,2,3,4,5,8,9; 1,2,3,5,6,8,9; 1,2,5,6,7,8,10; etc  
 1,2,3,4,5,10; 1,2,3,4,6,8; etc

**2. Weight of a cluster:** Weights are used to order the clusters according to their frequency of occurrence in the code. This helps in choosing amongst the clusters whenever there is an area limitation. After allocating area for the clusters with more weight, the others can be thrown to the on-chip micro processor core.

Parameters affecting weights:

**1. Overall frequency of the cluster**

This parameter is taken as param1. Param1 is the frequency of the cluster i.e. the number of times it will be used through the entire code. For e.g. if a cluster occurs in a zone with frequency 3 and zone with frequency 5, then its overall frequency is 8. For if-then, frequency is taken to be 0.5. Param1 for clusters inside loop whose limit is not known is greater than param1 for a non-loop code.

**2. Number of inputs and outputs of the cluster with weights assigned (any off-chip access requires 10 times as much power as on-chip access).**

When decision cannot be made using param1, this parameter known as param2 is taken into account. For param2, overall number is  $N(\text{internal}) + 10N(\text{external})$ .  $N(\text{internal})$  is the number of on-chip memory accesses given in terms of bits.  $N(\text{external})$  is the number of off-chip memory accesses.

**3. Number of instructions in the cluster with weights assigned.**

For param3, overall number is the sum of the number of each type of instruction weighted by their respective time of execution on an UltraSparc chip.

**Description of the function:**

**Input:** zone structure

**Output:**

- Cluster list  
 Struct cluster  
 {  
     list of instructions;  
     number of instructions;  
     number of occurrences;  
     list of occurrences;  
     weight of the cluster;  
 }  
 cluster\_list[];
- Three new fields added to the instruction structure
  - To specify whether it's been clustered
  - Cluster index to which it belongs
  - Bitstream to denote how it's present in the cluster

Eg: cluster\_list[5] = a,b,c,d. Instruction stream: A,b,d,f,g

- Instr = A;
- Instr → flag = clustered;
- Instr → index = 5;
- Instr → bs = 1101

The entire function is implemented in three stages:

- Initializing the cluster list
- Updating the cluster list
- Adjusting the cluster list

**Initializing the cluster list:** The cluster list is initialized with clusters of length 5 or can be taken to be empty. Function to initialize the cluster list with clusters of length L has been described here.

Function name: Initial

Input:

List of instructions in the 3 most frequent zones.

Output:

List of clusters each of length L which occur most frequently in the above list of instructions.

Description of the algorithm:

The algorithm parses through the .s file and initializes a 2-D array. The x-dimension as well as the y-dimension has the instructions found in the .s code as their indices. Each of the cells represents the number of occurrences of both the instructions in a sequence.

Eg:

	ld	st	add
ld	N(ld,ld)	N(ld,st)	
st	N(st,ld)		
add			

**Updating the cluster list:** Parsing of zones is done here.

The zones are parsed 6 times on the whole. For each parse, a temporary sequence (TS) of instructions starting from the current instruction and length (L) equal to the cluster with which it is currently being matched is created.

**Parse 1:** An exact match is required for TS to be a part of the cluster. This is done for all L's. Upon an exact match, no further parsing is done for the instructions involved.

**Parse 2:** A match with exactly 1 difference is required for TS to be a part of the cluster.

For L = 10, the difference has to be at any one end of TS.

**Parse 3:** A match with exactly 2 differences is required for TS to be a part of the cluster.

For L = 10, the differences have to be at any one end of TS.

For L = 9, one difference can be in the middle of TS

**Parse 4:** A match with exactly 2 differences is required for TS to be a part of the cluster. This is done only for L = 8,9 and 10

For L = 10, the differences have to be at any one end of TS.

For L = 9, one difference can be in the middle of TS

For L = 8, two differences can be in the middle of TS

**Parse 5:** A match with exactly 2 differences is required for TS to be a part of the cluster. This is done only for L = 10

For L = 10, the differences have to be at any one end of TS.

**Parse 6:** This is done to form new clusters. Any instruction that is ungrouped is considered here. If the length of the ungrouped instruction is equal to 4 or 5, then it is named as a new cluster. If length = 1,2 or 3 then the instructions are left ungrouped.

Updating the cluster list is finished when all the zones have been parsed.

After updating, the cluster structure will look like this.

```
Cluster_list[1]
{
    List of instructions = a,b,c,d,e,f
    Number of instructions = 6
    List of occurrences: Zone [1][16] (First zone 16th instruction), length
                        Zone [1][25], length
    Weight of the cluster: frequency (zone1)
}
```

**Adjusting the cluster list:** The ungrouped instructions are taken into account here. They are either done using a cluster that is not currently used. If the above is not possible, then the instructions are executed using the on-chip microprocessor.

**Assumptions:** An initial cluster length of five is assumed. This number is selected because it is not too big to cause a significant impact on hardware, but big enough to get some speedup through hardware implementation. The maximum length of a cluster is taken to be 10 as clusters bigger than that will be hard to find in the assembly code.

**Advantages:**

1. More efficient clustering

2. Lesser use of the general-purpose processor.
3. No effect of the control structure on the clustering algorithm.

#### Disadvantages:

1. More complexity in hardware implementation due to enable bit usage.

#### Pseudo code:

Function cluster(zone)

```
{
    /* Initialize the cluster list */

    /* Clusters are extracted from all the zones */
    while (all zones not processed)
    {
        /* assign zone → current zone */
        /* Update the cluster list .....call update(zone,cluster_list) */
        /* Have a parsing algorithm to parse the whole tree (current_zone =
current_zone → next) */
    }

    /* Separate instructions are taken care of now. They are either made part of the
existing clusters or given to the general-purpose processor */
    while (all zones not processed)
    {
        /* assign zone → current zone */
        /* Adjust the cluster list .....call adjust(zone,cluster_list) */
        /* Have a parsing algorithm to parse the whole tree (current_zone =
current_zone → next) */
    }
}
```

function initialize(cluster\_list)

```
{
    Input: node_list → Single linked list with nodes defined above as nodes
    Output: cluster_list → single linked list with cluster as nodes.

    /* This can be done either by taking the first five instructions of the first zone or
by parsing the whole code for most frequently occurring candidates */
```

Data structures used:

1. Instruction

```

typedef struct node
{
    op operation;
    char* opr1;
    char* opr2;
    char* opr3;
    int visited; /* 1 if visited, else 0 */
    int operand_count; /* how many operands does the instruction have */
    struct node* prev;
    struct node* next;
    int f_flag; /* 1 if function */
    int label_flag;
    int is_label_operand;
    int label_no;
    char* function_name;
    int original; /* 1 if original else 0 */
}
nnode, *node_list;

```

2. Cluster, as described in the clusterise function.

3. Intermediate table

Struct table\_element

```

{
    int x_opcodes[];
    int y_opcodes[];
    int frequency;
}
table[N][N];

```

1. Create a single linked list node\_list\_temp containing only the opcode indices as nodes.
2. Let total number of opcodes be N1.
3. Allocate memory to the 2-D array "table" to store N1xN1 elements.
4. For I = 2 to 5 do steps 5,6,7,8,9 and 10
5. Parse through node\_list\_temp and update "table".
- /\* Consider only those opcodes present in the indices of "table" \*/
6. Consider the first MI largest entries in the "table" and index them from (N(I-1)+1) to (N(I-1)+MI=NI).
7. Replace the indices in node\_list\_temp by the newly formed indices wherever the corresponding group of two opcodes are found.
8. Free the space allocated to "table"
9. Update the final\_cluster\_list with the above MI list of clusters.
10. if (I != 5) Reallocate memory to the 2-D array "table" to store MIxMI elements.
11. return (final\_cluster\_list);

}

function update(cluster\_list, zone)

```

{
    instructions *instr;
    /* Point temp = instr to the first instruction of the zone */
    /* repeat the following steps till the end of zone is reached */
    temp = instr;
    for I = 1 to no. of clusters
    {
        l = cluster_list[I] → number of instructions;
        /* Set first l instructions of instr equal to first l instructions of zone */
        /* check for perfect match between instr and cluster_list[I] → list of
instructions */
        /* If perfect match, then instr = instr + l, set the flag on all the instructions
in the zone to grouped, exit the loop */
        }
        /* if flag(temp) != grouped, then instr = instr + l */

        /* Point instr to the first instruction of the zone */
        /* repeat the following steps till the end of zone is reached */
        while (instr != grouped)
        {
            instr = instr + l
        }
        temp = instr;
        for I = 1 to no. of clusters
        {
            l = cluster_list[I] → number of instructions;
            /* Set first l instructions of instr equal to first l instructions of zone */
            /* check for match with one difference between instr and
cluster_list[I] → list of instructions */
            /* If one difference at the end, then instr = instr + l - 1, set the flag on all
the instructions in the zone to grouped, exit the loop */
            /* If one difference not at the end, then instr = instr + l, set the flag on all
the instructions in the zone to grouped, exit the loop */
            }
        /* if flag(temp) != grouped, then instr = instr + l */
    }
}

```

.....

**Example:**

The list of zones that are generated for the example explained in the "zonise" code has been used here. Initial list has been taken to be empty for simplicity.

Zone1:

```

st      %g0, [%fp-20]
st      %g0, [%fp-24]
st      %g0, [%fp-28]
st      %g0, [%fp-32]
st      %g0, [%fp-36]
st      %g0, [%fp-40]
st      %g0, [%fp-44]
st      %g0, [%fp-48]
mov     1, %o0
st      %o0, [%fp-20]
ld      [%fp-20], %o0
cmp     %o0, 9

```

Cluster list after parsing: c1: {st, st, st, st, mov, st}

Zone after parsing: c1, c1, (ld,cmp)

Zone 2:

```

ld      [%fp-20], %o0
add     %o0, 1, %o1
st      %o1, [%fp-20]
ld      [%fp-20], %o0
ld      [%fp-24], %o1
cmp     %o0, %o1

```

Cluster list after parsing: c2: {ld, add, st, ld, ld, cmp}

Zone after parsing: c2

Zone 3:

```

ld      [%fp-44], %o0
add     %o0, -8, %o1
st      %o1, [%fp-44]
ld      [%fp-44], %o0
mov     %o0, %o1
sll     %o1, 3, %o2
sub     %o2, %o0, %o0
st      %o0, [%fp-44]

```

.LL5:

```
ld    [%fp-20], %o0
add   %o0, 1, %o1
st    %o1, [%fp-20]
ld    [%fp-20], %o0
cmp   %o0, 9
```

Cluster list after parsing: c3: {mov, sll, sub, st}  
Zone after parsing: c2, c3, c2

Zone 4:

```
mov   23, %o0
st    %o0, [%fp-48]
```

Cluster list after parsing:  
Zone after parsing: c3

Zone 5:

```
mov   25, %o0
st    %o0, [%fp-40]
```

Cluster list after parsing:  
Zone after parsing: c3

Zone 6:

```
ld    [%fp-36], %o0
add   %o0, 5, %o1
st    %o1, [%fp-36]
ld    [%fp-40], %o0
add   %o0, 4, %o1
st    %o1, [%fp-40]
```

.LL2

```
ret
restore
```

Cluster list after parsing: c4: {add, st, ret, restore}  
Zone after parsing: c2, c4

Zone 7:

```
mov   19, %o0
st    %o0, [%fp-44]
```

Cluster list after parsing:

Zone after parsing: (mov, st)

Zone 8:

```
ld    [%fp-44], %o0
mov    %o0, %o2
sll    %o2, 2, %o1
add    %o1, %o0, %o1
sll    %o1, 2, %o0
st     %o0, [%fp-44]
ld     [%fp-44], %o0
add    %o0, -7, %o1
st     %o1, [%fp-44]
```

.LL14:

```
ld     [%fp-28], %o0
cmp    %o0, 8
```

Cluster list after parsing:

Zone after parsing: (ld, mov),c3, c2

Zone 9:

```
ld     [%fp-44], %o0
add    %o0, -23, %o1
st     %o1, [%fp-44]
```

Cluster list after parsing:

Zone after parsing: c2

Zone 10:

```
ld     [%fp-44], %o0
add    %o0, 17, %o1
st     %o1, [%fp-44]
ld     [%fp-20], %o0
ld     [%fp-44], %o1
call   .umul, 0
nop
st     %o0, [%fp-20]
```

Cluster list after parsing:

Zone after parsing: c2, (call umul, st)

Zone 11:

```

mov  9, %o0
st    %o0, [%fp-40]
ld    [%fp-28], %o0
cmp   %o0, 0

```

Cluster list after parsing: c5: {mov, st, ld, cmp}

Zone after parsing: c5

Zone 12:

```

mov  10, %o0
st    %o0, [%fp-32]
ld    [%fp-40], %o0
ld    [%fp-48], %o1
add   %o0, %o1, %o0
st    %o0, [%fp-36]
.LL10:
ld    [%fp-28], %o0
add   %o0, -14, %o1
st    %o1, [%fp-28]
ld    [%fp-44], %o0
mov   %o0, %o1
sll   %o1, 3, %o0
mov   7, %o1
sub   %o1, %o0, %o0
st    %o0, [%fp-28]

```

Cluster list after parsing:

Zone after parsing: c5, c2, c2, c3

Zone 13:

```

ld    [%fp-20], %o0
ld    [%fp-44], %o1
cmp   %o0, %o1

```

Cluster list after parsing:

Zone after parsing: c2

Assuming none of these zones occur in parallel,

List of clusters required in hardware: c1 (2), c2 (2), c3 (1), c4 (1) and c5 (1).

List of reconfigurations: Zone 1 and Zone 3 are available in hardware at the start.

.....

## Relevant Work

### 1. Chameleon Systems

#### 1.1 Tool suite supports reconfigurable processor

SUNNYVALE, Calif. — Claiming to provide the first development environment for a reconfigurable signal processor, Chameleon Systems this week will announce C-Side (Chameleon Systems Integrated Design Environment), a tool set for its CS-2112 reconfigurable communications processor (RCP).

C-Side uses a combined C language and Verilog flow to map algorithms into the chip's reconfigurable processing fabric (RPF). Chameleon introduced the CS2112 earlier this year. The chip won't be in full production until the fourth quarter, but the tools are being made available now so designers can start application development.

"So far as we are aware, we are the only reconfigurable processor supplier that has taken the path of providing tools to the customer so they can build their own custom algorithms," said Tim Erjavec, vice president of marketing for Chameleon.

The tool set lets users map algorithms into the CS2112's RPF, which includes four slices, each of which can be reconfigured independently. Each slice has three tiles, and each tile includes seven 32-bit data-path units (DPUs), two multipliers, four local-store memories and a control logic unit.

C-Side also provides programming and debugging for the chip's ARC processor core, which provides top-level management of chip resources for a given reconfiguration. Offering a higher level of granularity than FPGAs, the CS2112 allows reconfigurability at the microarchitectural level. That lets users reload slices with algorithms without having to work at a bit level. The architecture purportedly offers more flexibility than standard DSPs because CS2112 users can specify both functions and interconnect between functional units.

The C-Side tool set includes an optimized GNU compiler for the ARC core, an optimized Verilog synthesizer for the RPF, an interactive floor planner, an instruction-set simulator and a unified debug environment for the ARC core and the RPF. It comes with a development board that includes a single CS2112 along with memory, a PCI interface and a programmable I/O module.

#### Familiar methodology

Users start the design process with a C language description of their signal processing algorithms. They write a testbench and run an initial simulation on the ARC core. Then users identify data-path-intensive blocks, called kernels, which are targeted to the RPF. Users create functions for those blocks in Chameleon's assembly-like design entry language, which generates standard Verilog descriptions.

Raj Karamchedu, Chameleon's senior marketing manager, said the company plans to allow design entry from such tools as Matlab or SPW. That will let users draw data-flow diagrams in lieu of writing code. Until then, users must recapture portions of their C program in Verilog, because there's no direct translation between the two.

To program the DPUs, users wire together library elements ranging from adders and multipliers to FIR filters and Viterbi decoders. Those blocks are currently fixed, but Karamchedu said Chameleon is working to parameterize them. Control logic is programmed with Verilog state machines.

Chameleon's V2B (Verilog to bits) synthesizer takes the Verilog descriptions all the way to a placed-and-routed bit stream. While placement and routing are automatic, C-Side includes an interactive floor planner that lets users view a complete map of the chip's slices and tiles, get estimates of routability and manually rearrange the placement.

As a given kernel is programmed, the user brings the bit stream back into the ARC testbench, replacing the original fixed-point C model. A utility called eBios inserts calls in the testbench that allocate slices, load the kernel into slices, activate the configuration and execute the kernel. The testbench that includes the eBios calls is compiled using a GNU C compiler and is downloaded into the CS2112 development board.

While the Verilog kernel descriptions can be verified on any standard Verilog simulator, C-Side also includes ChipSim, an instruction-set simulator for the Chameleon architecture. It models the entire chip, including the ARC core and the RPF, using the GNU Debugger front end. Users can view all memories and registers on the CS2112, including the ARC core, and can single-step to debug the kernel.

#### 'Real hardware debug'

A feature called VCD Dump lets users capture debug information from the internal nodes and registers of the CS2112 from both the development board and ChipSim. "Either way you get real hardware debug," Karamchedu said.

The development board also lets designers connect multiple RCPs to other devices in the system using the PCI bus or programmable I/O pins.

Karamchedu acknowledged a "learning curve" for designers unfamiliar with reconfigurable logic but said that a typical design cycle for a basestation, now 12 to 18 months, can probably be reduced to six to eight months using C-Side and the CS2112.

C-Side sells for \$25,000 on Solaris platforms. The development board sells for \$5,000.

Referenced from <http://www.eetimes.com/story/OEG20010813S0079>

### 1.2.Chameleon Systems First to Deliver Development Environment for Reconfigurable Communication Processor

SAN JOSE, Calif. — August 20, 2001 — Chameleon Systems, Inc. today announced it has begun shipping a complete development environment for the CS2112 Reconfigurable Communication Processor (RCP) — the company's flagship product announced in May of last year. The development environment, comprising Chameleon's C-SIDE software tool suite and CT2112SDM development kit, enables customers to develop and debug communication and signal processing systems running on the RCP. Chameleon has completed its development program and is now shipping tools and development boards to its customers.

"The definitive value of reconfigurable signal processing technology derives from the ability to design and implement custom algorithms on custom processor architectures," said Chuck Fox, president and CEO of Chameleon. "For any given application, the structure of the processor architecture is critical to both the performance and function of the algorithms it runs. A truly reconfigurable processor is one that allows the designer to create an architecture with the optimum mix of parallel and pipelined processing elements, memory blocks and I/O ports for their algorithm. Chameleon Systems is the first reconfigurable processor company to deliver to its customers a complete and dynamic flexibility for both architectural and algorithmic development."

The RCP's development environment helps overcome a fundamental design and debug challenge facing communication system designers. In order to build sufficient performance, channel capacity, and flexibility into their systems, today's designers have been forced to employ an amalgamation of DSPs, FPGAs and ASICs, each of which requires a unique design and debug environment. The RCP platform was designed from the ground up to alleviate this problem: first by significantly exceeding the performance and channel capacity of the fastest DSPs; second by integrating a complete SoC subsystem, including an embedded microprocessor, PCI core, DMA function, and high-speed bus; and third by consolidating the design and

debug environment into a single platform-based design system that affords the designer comprehensive visibility and control.

#### **Complete Tool Suite**

The C-SIDE software suite includes tools used to compile C and assembly code for execution on the CS2112's embedded microprocessor, and Verilog simulation and synthesis tools used to create parallel datapath kernels which run on the CS2112's reconfigurable processing fabric. In addition to code generation tools, the package contains source-level debugging tools that support simulation and real-time debugging.

The development board includes a single CS2112 chip on a PCI form-factor board which allows for hardware accelerated development, debugging, and testing of complex physical layer systems of wireless communication protocols such as UMTS and cdma2000 designed for the RCP.

#### **Familiar Design Approach**

Chameleon's design approach leverages the methods employed by most of today's communications system designers. The designer starts with a C program that models signal processing functions of the baseband system. Having identified the dataflow intensive functional blocks, the designer implements them in the RCP to accelerate them by 10-100X. The designer creates equivalent functions for those blocks, called kernels, in Chameleon's reconfigurable assembly language-like design entry language. The assembler then automatically generates standard Verilog for these kernels that the designer can verify with commercial Verilog simulators. Using these tools, the designer can compare test bench results for the original C functions with similar results for the Verilog kernels.

In the next phase, the designer synthesizes the Verilog kernels using Chameleon's synthesis tools targeting Chameleon technology. At the end, the tools output a bit file that is used to configure the RCP. The designer then integrates the application level C code with Verilog kernels and the rest of the standard C function. Chameleon's C-SIDE compiler and linker technology makes this integration step transparent to the designer. At this stage the designer can use Chameleon's full-chip simulator (ChipSim) to verify how the overall C code is performing with respect to system specifications.

#### **Processor-like Debug**

Through all phases of the Chameleon flow, the designer exercises the system and test bench design with real data vectors. The CS2112 development environment makes all chip registers and memory locations accessible through a development console that enables full processor-like debugging, including features like single-stepping and setting breakpoints. Designers can perform all of these debugging procedures within the ChipSim simulator, or use the CT2112SDM development board with hardware implementation on the actual chip. Before actually productizing the system, the designer must often perform a system-level simulation of the data flow within the context of the overall system. Chameleon's development board enables the designer to connect multiple RCPs to other devices in the system using the PCI bus and/or programmable I/O pins. This helps prove the design concept, and enables the designer to profile the performance of the whole base-station system in a real-world environment.

#### **Pricing and Availability**

The C-SIDE Software tool suite (Solaris, floating license) is priced at of \$25,000, and includes two years of maintenance, software updates and support. The Chameleon CT2112SDM Development Kit is priced at \$5,000. Production shipments are available now.

<http://www.chameleonsystems.com>

For more information contact:  
Marc Koltun, VitalCom Public Relations

(408) 240-3408

[marc@chameleonsystems.com](mailto:marc@chameleonsystems.com)

Referenced from <http://www.chameleonsystems.com/press/20Aug2001.html>

### 1.3 White paper reports of Chameleon Systems Inc.

#### Wireless Base Station Design Using

#### Reconfigurable Communications Processors

As the communications market continues its explosive growth and rapid rate of change, equipment vendors struggle with the conflicting goals of performance, flexibility, low cost and fast time-to-market. Traditional processing approaches such as DSPs, ASICs, ASSPs and FPGAs all force the designer to sacrifice at least one of these key parameters. A new class of processor from Chameleon Systems, the Reconfigurable Communications Processor (RCP), enables designers to meet all these goals simultaneously for multi-channel, data-processing intensive applications.

Full paper is in the attachment.

### 2.PP: A Path Profiling Tool

Glenn Ammons, Tom Ball, James Larus

A program path records a program's control transfers through a sequence of consecutively executed basic blocks. Although a program's execution traces a single path, practicality demands this path be broken into shorter, more manageable path segments. The difficulty is that the number of potential paths through a program with loops is unbounded, which makes individual paths difficult to identify and name. On the other hand, a complete path can be assembled from shorter subpaths, such as Ball and Larus's acyclic paths. Paths are useful for two reasons. First, they concisely capture the execution history of many instructions, and so record a program's dynamic control flow. The set of (acyclic) paths executed by a program, its path spectra, compactly describes much of the program's dynamic behavior. Second, the control locality of paths is even more pronounced than code locality in a program as whole. Code locality is typified by the 80-20 rule—the observation that 80% of a program's execution occurs in only 20% of its code. If programs are viewed as a collection of paths, the 80-20 rule becomes the 100-0 rule, since programs execute only a miniscule fraction of the possible paths through the nearly infinite maze of their flow graph. The 80-20 rule reappears, however, within the domain of executed paths, as programs spend most of their time in a small number of "hot paths."

PP is a fast path profiler. A path profiler associates event counts with paths through programs. For example, the events can be simply the beginnings of new paths, in which case the profile just counts the number of times each path executes, or the events can be hardware events like data cache misses, in which case the profile counts the number of times each path suffered a cache miss.

PP does two different kinds of path profiling: flow profiling and context profiling. In flow profiling, the paths are acyclic, intraprocedural paths through control flow graphs. In context profiling, the paths are paths in the call graph: events are associated with nodes in a data structure called the calling context tree, which is a bounded representation of the dynamic call tree.

PP is built on EEL, a library for editing executable files. EEL currently runs on SPARC-based machines (SunOS and Solaris). The machine-specific code in EEL and QPT2 is collected in a few files. Porting to a new machine requires a couple months of effort.

HPB (Hot Path Browser) is a Java tool that can display the paths that a program executes. It reads the output of PP and displays a program's executed path in several linked windows.

### Documentation

- The algorithms in PP are described in: Thomas Ball and James Larus, *Efficient Path Profiling*, MICRO-29, December 1996.
- Glenn Ammons, Thomas Ball, and James Larus, *Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling*, PLDI '97, June 1997.

## Obtaining PP

PP is distributed along with EEL, which is available as part of WARTS.

PP is distributed with the full source and a small amount of documentation. PP is copyrighted by James Larus and Glenn Ammons and is distributed under the terms of the WARTS license. A copy of the license is available on <ftp.cs.wisc.edu> in [~ftp/pub/warts/license.ps](ftp://pub/warts/license.ps), or can be obtained by contacting me at the address below.

James Larus  
Computer Sciences Department  
1210 West Dayton Street  
University of Wisconsin  
Madison, WI 53706  
[larus@cs.wisc.edu](mailto:larus@cs.wisc.edu)  
(608) 262-9519

### 3. Speeding up program execution using reconfigurable hardware and a hardware function library

Jain, S.; Balakrishnan, M.; Kumar, A.; Kumar, S.

VLSI Design, 1998. Proceedings., 1998 Eleventh International Conference on , 1998

Page(s): 400-405

#### Abstract:

This paper describes a co-design environment which follows a new approach for speeding up compute intensive applications. The environment consists of three major components. First, a target architecture consisting of a uniprocessor host and a board with dynamically reconfigurable FPGAs and memory modules; second, a library of functions pre-synthesized for hardware or software implementation; and third, a tool which takes as input an application described in C and partitions it into hardware and software parts at functional granularity using information obtained by profiling the application. An important feature of the partitioning tool is a new efficient heuristic specifically suited for the architecture with reconfigurable hardware.

Full paper attached separately.

### 4. SCORE (Stream Computations Organized for Reconfigurable Execution ) at Berkeley

<http://brass.cs.berkeley.edu/SCORE/>

*"The challenge is to find the right computational abstractions to characterize the family of reconfigurable devices we can envision, expose a uniform view to the programmer, and represent the computation in a manner which a wide-range of hardware implementations can exploit efficiently. The BRASS project is developing a stream-oriented computational model to address this issue, providing an abstract view of the reconfigurable hardware, which exposes its strengths while abstracting the actual composition of physical resources. We call this stream-oriented computational model SCORE. The key concept in this model is that a computation is broken up into compute pages. Compute pages are linked together in a data-flow manner with streams. A run-time OS manager allocates and schedules pages at run-time for both for computations and memory."*

Pleiades is exploring reconfiguration of coarser-grain, application-specific building blocks with an emphasis on low-power computations.

([http://bwrc.eecs.berkeley.edu/Research/Configurable\\_Architectures/](http://bwrc.eecs.berkeley.edu/Research/Configurable_Architectures/))

*"The Pleiades project at UC Berkeley seeks to achieve ultra-low power high-performance multimedia computing through the reconfiguration of heterogeneous system modules. Achieving high-energy efficiency requires the elimination of the waste that typically dominates the energy consumption in general-purpose programmable engines. Providing programmability at just the right granularity (instruction, functional module, data path or gate) makes it possible to eliminate virtually all overhead, while making it further*

*possible to exploit other energy reducing techniques, such as parallelism, pipelining and dynamic voltage scaling. "*

#### Papers:

Jan M. Rabaey, "Reconfigurable Processing: The Solution to Low Power Programmable DSP ,"  
Proceedings 1997 ICASSP Conference, Munich, April 1997.

This paper presents an approach to low power programmable DSP that is based on the dynamic reconfiguration of hardware modules.

Suet-Fai Li, Marlene Wan, Jan Rabaey, "Configuration Code Generation and Optimization for Low-Energy Reconfigurable DSPs", *Proceedings of SIPS99*

**Abstract** - In this paper we describe a code generation and optimization process for reconfigurable architectures targeting digital signal processing and wireless communication applications. The ability to generate efficient and compact code is essential for the success of reconfigurable architectures. Otherwise, the overhead of reconfiguring could easily become the system bottleneck. Our code generation process includes the evaluation a set of tradeoffs in system design, software engineering as well as usage of a set of local and global optimization techniques. By doing so we are able to achieve results of significantly lower overhead.

#### 5. Instruction Generation and Regularity Extraction For Reconfigurable Processors

Philip Brisk, Adam Kaplan, Ryan Kastner, Majid Sarrafzadeh

Computer Science Department

University of California, Los Angeles

Los Angeles, CA 90095

{philip, kaplan, kastner, majid}@cs.ucla.edu

#### ABSTRACT

The increasing demand for complex and specialized embedded hardware must be met by processors which are optimized for performance, yet are also extremely flexible. In our work, we explore the tradeoff between flexibility and performance in the domain of reconfigurable processor design. Specifically, we seek to identify regularly occurring, computation-heavy patterns in an application or set of applications. These patterns become candidates for hard-logic implementation, potentially embedded in the flexible reconfigurable fabric as special optimized instructions. In this work we present an extension to previous work in instruction generation: an algorithm that identifies parallel templates. We discuss the advantages of parallel templates, and prove the correctness of our algorithm. We introduce an All-Pairs Common Slack Graph (APCSG) as an effective tool for parallel template generation. Finally, we demonstrate the effectiveness of our algorithm on several applications' dataflow graphs, reducing latency on average by 51.98%, without unreasonably increasing chip area.

Full paper is in the attachment.

#### 6 Computer Architecture Laboratory at Delft University of Technology, Netherlands

6.1 Marnix Arnold, Henk Corporaal, "Automatic Detection of Recurring Patterns" Computer Architecture Laboratory, Department of Electrical Engineering Delft University of Technology, Seventh International Workshop on Hardware/Software Co-Design (CODES'99), Rome Italy, May 1999.

6.2 Marnix Arnold, "Matching and Covering with Multiple Output Patterns", Computer Architecture Laboratory, Department of Electrical Engineering Delft University of Technology Technical report no. 1-68340-44/1999/01, Delft University of Technology, January 1999

6.3 Marnix Arnold, Henk Corporaal, "Instruction Set Synthesis Using Operation Pattern Detection", Computer Architecture Laboratory, Department of Electrical Engineering Delft University of Technology, Fifth Annual Conf. of ASCI, Heijen, The Netherlands, June 1999.

## Efficient Path Profiling

Thomas Ball  
Bell Laboratories  
Lucent Technologies  
tball@research.bell-labs.com

James R. Larus\*  
Dept. of Computer Sciences  
University of Wisconsin-Madison  
larus@cs.wisc.edu

### Abstract

A path profile determines how many times each acyclic path in a routine executes. This type of profiling subsumes the more common basic block and edge profiling, which only approximate path frequencies. Path profiles have many potential uses in program performance tuning, profile-directed compilation, and software test coverage.

This paper describes a new algorithm for path profiling. This simple, fast algorithm selects and places profile instrumentation to minimize run-time overhead. Instrumented programs run with overhead comparable to the best previous profiling techniques. On the SPEC95 benchmarks, path profiling overhead averaged 31%, as compared to 16% for efficient edge profiling. Path profiling also identifies longer paths than a previous technique, which predicted paths from edge profiles (average of 88, versus 34 instructions). Moreover, profiling shows that the SPEC95 train input datasets covered most of the paths executed in the ref datasets.

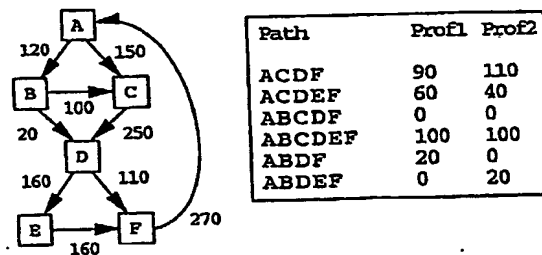


Figure 1. Example in which edge profiling does not identify the most frequently executed paths. The table contains two different path profiles. Both path profiles induce the same edge execution frequencies, shown by the edge frequencies in the control-flow graph. In path profile Prof1, path ABCDEF is most frequently executed, although the heuristic of following edges with the highest frequency identifies path ACDEF as the most frequent.

## 1 Introduction

Program profiling counts occurrences of an event during a program's execution. Typically, the measured event is the execution of a local portion of a program, such as a routine or line of code. Recently, fine-grain profiles—of basic blocks and control-flow edges—have become the basis for profile-driven compilation, which uses measured frequencies to guide compilation and optimization.

\*This research supported by: Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550; NSF NYI Award CCR-9357779, with support from Hewlett Packard, Sun Microsystems, and PGI; NSF Grant MIP-9225097; and DOE Grant DE-FG02-93ER25176. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U. S. Government.

One use of profile information is to identify heavily executed paths (or traces) in a program [Fis81, Ell85, Cha88, YS94]. Unfortunately, basic block and edge profiles, although inexpensive and widely available, do not always correctly predict frequencies of overlapping paths. Consider, for example, the control-flow graph (CFG) in Figure 1. Each edge in the CFG is labeled with its frequency, which normally results from dynamic profiling, but in the figure is induced by both path profiles in the table. A commonly used heuristic to select a heavily executed edge out of a basic block [Cha88], which identifies path ACDEF. However, in path profile Prof1, this path executed only 60 times, as compared to 90 times for path ACDF and 100 times for path ABCDEF. In profile Prof2, the disparity is even greater although the edge profile is exactly the same.

This inaccuracy is usually ignored, under the assumption that accurate path profiling must be far more expensive than basic block or edge profiling. Path profiling is the ultimate form of control-flow profiling, as it uniquely deter-

mines both basic block and edge profiles, although the converse does not hold, as Figure 1 shows. Also, the number of blocks or edges in a program is finite and linear in the program's size, but a program with loops offers an unbounded number of potential paths. Considering only acyclic paths bounds this set, but, in the worst case, its size is still exponential in the program's size.

This paper shows that accurate profiling is neither complex nor expensive. It describes a new and efficient technique for *path profiling*. Our algorithm places instrumentation that accurately determines dynamic execution frequency of control-flow paths in a routine. The instrumentation is not only simple and low-cost, but it is placed in a way that minimizes its overhead. Remarkably, although path profiling collects far more information than block or edge profiling, its overhead can be lower and is usually comparable—on the SPEC95 benchmarks, path profiling's average overhead is 31%, while efficient edge profiling's overhead is 16%.

Efficient path profiling opens new possibilities for program optimization and performance tuning. Instead of relying on heuristics, which fully predict only 38% of the executed acyclic paths in the SPEC95 benchmarks, profile-driven compilers can base their decisions on accurate measurements.

Another potential application of path profiling is software test coverage, which quantifies the adequacy of a test data set by profiling a program and reporting unexecuted statements or control-flow. Few, if any, coverage tools measure path coverage. Instead, tools rely on weaker criteria, such as statement or control-flow edge coverage. Edge profiling is less complete than path profiling, as shown in Figure 1, where the two path profiles cover different sets of paths yet induce the same edge profile. Besides an efficient algorithm for path profiling, this paper also presents measurements that show that most *routines* in a small sample of programs have few ( $< 3000$ ) potential paths, so that path coverage testing could be feasible for large portions of a program. On the other hand, the measurements also demonstrate the difficulty of developing test data sets, since the programs as a whole executed an average of 2696 paths (249–24414), as compared to the millions of potential paths identified by the path profiling algorithm.

### 1.1 Algorithm Overview

The essential idea behind the path profiling algorithm is to identify sets of potential paths with states, which are encoded as integers. Consider for a moment a routine without a loop. Upon entry to the routine, all paths are possible. Taking a conditional branch narrows the set of potential paths and corresponds to a transition to a new state. At the routine's exit, the final state represents the single path taken

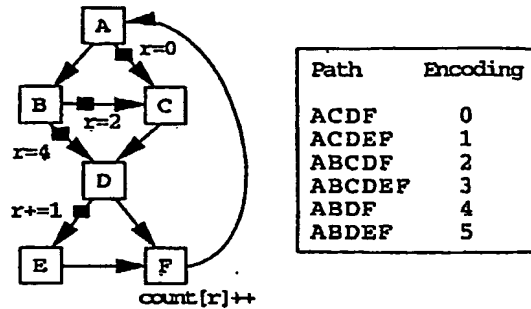


Figure 2. Path profiling instrumentation. Each path from A to F produces a unique state in register  $r$ , which indexes an array of counters in  $F$ .

through the routine. This paper presents an efficient algorithm that:

- Numbers final states from  $0 \dots n - 1$ , where  $n$  is the number of potential paths in a routine. With this compact numbering, a final state can directly index an array of counters.
- Places instrumentation so that transitions need not occur at every conditional branch.
- Assigns states so that transitions can be computed by a simple arithmetic operation, without an explicit state transition table or memory reference.
- Transforms a control-flow graph containing loops or huge numbers of potential paths into an acyclic graph with a limited number of paths.

Figure 2 illustrates the technique. Edges labeled by small squares contain instrumentation, which updates the state in register  $r$ . The loop contains six unique paths, and each one computes a different value for  $r$ , as shown in the table. At the end of the loop body (block F), register  $r$  holds the index to increment an array of counters.

### 1.2 Extensions

The algorithm in this paper can be easily extended in several ways. First, instead of intraprocedural profiling, it could be applied to a program's call graph, to record call paths. An interesting complication is indirect calls, which require a dynamic data structure to record calls along edges that are not in the call graph.

Also, instead of just counting the number of times a path executes, the profiling algorithm can easily accumulate a metric for a path. Some processors provide accessible counters for metrics such as the number of processor cycles,

stalls, cache misses, or page faults. A minor change to the path profiling code could increment a path's counter by the change in a counter over the path.

### 1.3 Paper Overview

The path profiling algorithm instruments a program to record paths with low run-time overhead. The algorithm uses previous results on efficient profiling and tracing [BL94] and efficient event counting [Bal94] to determine which edges to instrument. The contribution of this paper is combine these algorithms, apply them to a new problem, and develop a new algorithm to compute an update constant for each instrumented edge. The algorithm ensures that each distinct path generates a unique value. Furthermore, the path encoding is compact and minimal, so that the maximum value for any path is the number of unique paths through a CFG (minus one), as in Figure 2. A simple, linear-time algorithm achieves both goals (Section 3).

This paper only considers intra-procedural acyclic paths, which result from removing loop backedges before instrumentation (Section 4). This process produces a profile that does not capture paths that cross a backedge. However, acyclic path profiling counts the number of times that a loop iterates and records both paths into the first iteration and out of the last loop iteration. The same approach, of removing edges, can also limit the number of paths in complex routines, so that states can be represented as 32-bit integers. Even so, large routines can have too many states to use an array of counters. In this case, a hash table records paths that actually execute, so that the space overhead is proportional to the number of dynamic paths, rather than the number of potential static paths. The relatively high cost of hashing accounts for the higher overhead of path profiling, as compared to edge profiling.

We implemented the algorithms presented here in a profiling tool, *PP*, which uses the EEL library [LS95] to insert instrumentation into executable binaries (Section 5). This paper compares *PP* against QPT2, another profiling tool built with EEL, which uses an efficient edge profiling algorithm [BL94]. QPT2 usually incurred less overhead, but the two system were roughly comparable. Profiling the SPEC95 benchmarks, *PP*'s overhead averaged 31% (6–97%) while QPT2's overhead averaged 16% (2.6–53%) (Section 6). The measurements also compare profiled paths against paths predicted using edge profiles and show that for the SPEC95 benchmarks, which execute few unique paths, profiling identifies longer paths (an average of 7 CFG edges and 88 instructions, versus 5 edges and 34 instructions for predicted paths). Moreover, path profiling shows that the paths executed with the SPEC95 *train* dataset cover most of the dynamically executed instructions in the *ref* dataset, which suggests that path profiles could help improve

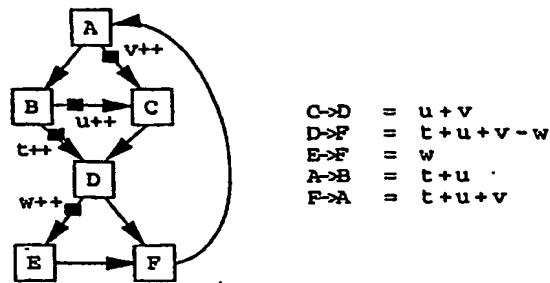


Figure 3. Instrumentation for edge profiling.

SPEC95 peak numbers.

## 2 Related Work

The path profiling algorithm, like previous work on efficient profiling and tracing techniques [BL94, Gol91], uses a spanning tree to determine a minimal, low-cost set of edges to instrument. For example, Figure 3 shows the control-flow graph from Figure 2 instrumented for edge profiling (the uninstrumented edges form a spanning tree). The same set of edges are instrumented in both cases. However, for edge profiling, each instrumented edge has its own counter (held in memory), which is incremented each time the edge executes. Figure 3 also shows how uninstrumented edges' counts are derived from recorded counts.

Path profiling produces a more detailed profile, although it instruments the same set of edges. Moreover, most path profiling instrumentation consists of register instructions, while every edge profiling instrumentation increments memory. In general, a path that executes  $N$  memory increments for edge profiling will execute  $N$  register initializations/adds plus one memory increment for path profiling. In practice, there are many procedures for which the number of potential paths is small (so arrays may be used) and path profiling incurs less overhead than edge profiling. However, there are procedures that have so many potential paths that a hash table must be used to store the profile.

Young and Smith used a limited form of program tracing to record paths for their branch correlation studies [YS94]. In a FIFO buffer, they recorded the last  $n$  branches, each of which consists of a basic block number and branch outcome. This technique is both more expensive than path profiling and also requires another level of indirection to associate a counter with a path, which consists of a sequence of block numbers. Unlike path profiling, this technique need not distinguish cyclic from acyclic paths since it truncates both at the FIFO boundary.

Bit tracing is another approach to path profiling. Bit tracing associates a 1-bit value with the outcome of each two-

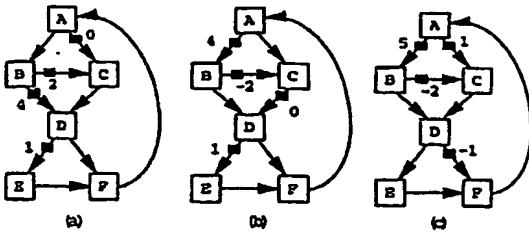


Figure 4. Three possible placements of instrumentation for the control-flow graph from Figure 1.

way branch [BL94, Bal96]. When a branch executes, instrumentation code appends a bit to a trace buffer that records branch outcomes. By recording multiple bits, the approach can be extended to multi-way branches. The contents of the buffer form an index into an array or as a hash value.

It is easy to see that bit tracing uses the minimal number of bits necessary to distinguish paths. For simple control-flow graphs, such as a chain of if-then-else statements, bit tracing, like our approach, produces a compact representation of paths. However, in general, bit tracing may not yield the most compact representations of paths possible. It is easy to construct examples for which the maximal path value under bit tracing is not minimal, no matter the choice of bit labellings. In the worst case, the number of entries in an array of counters may be twice our method.

In addition, bit tracing is likely to have higher run-time overhead than our approach. First, every predicate must be instrumented, whereas our approach allows flexibility in placing instrumentation to reduce overhead. Second, on most machines, the instrumentation to append to a bit string is more complex and slower than a register-to-register addition.

### 3 Path Profiling of DAGs

As described previously, path profiling tracks a path in a directed acyclic graph (DAG) by updating a register along certain edges of the DAG. This section shows how to compute the necessary updates, efficiently place instrumentation, and derive an executed path from the resulting profile.

The example in Figure 4 shows that many placements of instrumentation yield equivalent results. However, some placements incur less run-time overhead than others. For example, all three graphs in Figure 4 produce the same sum along any acyclic path from A to F. However, in graph (a), the largest number of instrumented edges on any path from A to F is two, while graphs (b) and (c) have up to four and three, respectively.

The path profiling algorithm first labels edges in a DAG

with integer values, such that each path from the entry to the exit of the DAG produces a unique sum of the edge values along that path (the *path sum*). However, placements from this step may have sub-optimal run-time overhead, as above.

In the next step, another algorithm [Bal94] improves this computation, by finding an equivalent computation that uses a minimal number of additions along DAG edges that are not in the DAG's spanning tree. In each graph in Figure 4, the uninstrumented edges (those without squares along them) form a spanning tree. Since a DAG may have many spanning trees, the algorithm has the freedom to place instrumentation along edges less likely to be executed.<sup>1</sup>

After reviewing the basic graph terminology in Section 3.1, this section describes the four basic steps to path profile a DAG:

1. Assign integer values to edges such that no two paths compute the same path sum (Section 3.2). This encoding is minimal.
2. Use a spanning tree to select edges to instrument and compute the appropriate increment for each instrumented edge (Section 3.3).
3. Select appropriate instrumentation (Section 3.4).
4. After collecting the run-time profile, derive the executed paths (Section 3.5).

#### 3.1 Terminology

For the remainder of this paper, unless otherwise noted, control-flow graphs (CFGs) have been converted into directed acyclic graphs (DAG) with a unique source vertex *ENTRY* and sink vertex *EXIT*. Section 4 shows how to transform an arbitrary CFG into a DAG, which can be path profiled. For technical reasons, the increment computation (Section 3.3) requires a "dummy" edge *EXIT* → *ENTRY* (although this creates an unexecutable cycle, the graph can still be treated as a DAG by ignoring this backedge).

An execution of a DAG produces an acyclic, directed path starting at *ENTRY* and ending at *EXIT*. The term *path* refers to an acyclic directed path, unless otherwise noted. Of course, a DAG may execute many times, as it may consist of a loop body or a procedure.

A spanning tree of a graph *G* is a subgraph that is a tree and contains all vertices of *G*. Edges in a spanning tree are bidirectional and need not follow the direction of graph

<sup>1</sup>This approach requires computing (or obtaining from a profile) a weight for each edge that statically approximates the edge's execution frequency. A maximum spanning tree of the graph, with respect to that weighting, maximizes the weight (execution frequency) of the uninstrumented edges. PP uses the same previously published, effective algorithm for statically computing a weighting as QPT [BL94].

```

foreach vertex  $v$  in reverse topological order {
  if  $v$  is a leaf vertex {
    NumPaths( $v$ ) = 1;
  } else {
    NumPaths( $v$ ) = 0;
    for each edge  $e = v \rightarrow w$  {
      Val( $e$ ) = NumPaths( $v$ );
      NumPaths( $v$ ) = NumPaths( $v$ ) + NumPaths( $w$ );
    }
  }
}

```

Figure 5. Algorithm for assigning values to edges in a DAG.

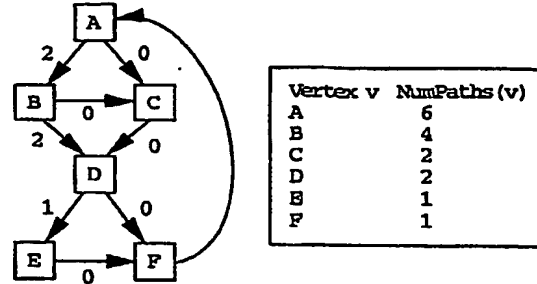


Figure 6. Control-flow graph from Figure 1, with values computed by the algorithm in Figure 5.

edges. If  $T$  is the set of spanning tree edges, then any graph edge not in  $T$  is a *chord* of the spanning tree.

For example, in the graph of Figure 2, vertex  $A$  is the *ENTRY* vertex and vertex  $F$  is the *EXIT* vertex. The unadorned graph edges comprise a spanning tree. The edges labeled by squares are chords of the spanning tree.

### 3.2 Compactly Representing Paths with Sums

The first step in path profiling is to assign a non-negative constant value  $Val(e)$  to each edge  $e$  in a DAG, such that the sum of values along any path from *ENTRY* to *EXIT* is unique. Furthermore, the path sums should lie in the range from 0 to the number of paths (minus one), so that the encoding is minimal.

The algorithm in Figure 5 computes such a  $Val$  relation by visiting vertices of the DAG in reverse topological order. This order ensures that all the successors of a vertex  $v$  are visited before  $v$  itself. Associated with each vertex  $v$  is a value  $NumPaths(v)$ , which records the number of paths from  $v$  to *EXIT*. The algorithm is simple. At vertex  $v$ , the algorithm visits all of  $v$ 's outgoing edges  $v \rightarrow w_i$ ,  $1 \leq i \leq n$ , and assigns the  $k^{th}$  outgoing edge the value:

$$Val(v \rightarrow w_k) = \sum_{i=1}^{k-1} NumPaths(w_i)$$

The following theorem proves the algorithm correct:

**Theorem 1** *Given a DAG, after the algorithm of Figure 5 visits vertex  $v$ ,  $NumPaths(v)$  is the number of paths from  $v$  to *EXIT* and each path from  $v$  to *EXIT* generates a unique value sum in the range  $0 \dots NumPaths(v) - 1$ .*

**Proof.** By induction on the height of a vertex in the DAG (i.e., the max number of steps to the sink vertex *EXIT*).

**Base Case:**  $v$  has height equal to zero (that is,  $v = EXIT$ ), so  $NumPaths(v) = 1$ . The theorem is trivially satisfied.

**Induction Step:** Show that the theorem holds for any vertex  $v$  of height  $H$  ( $H > 0$ ). All successors  $w_1 \dots w_n$  of  $v$  must have height less than  $H$  (because the graph is a DAG), so the theorem holds for all  $w_i$ . It is trivial to see that the number of paths from  $v$  to *EXIT* is  $\sum_{i=1}^n NumPaths(w_i)$ , which the algorithm computes. By the induction hypothesis, each path from  $w_k$  to *EXIT* generates a unique value sum in the range  $0 \dots NumPaths(w_k) - 1$ . Therefore, any path from  $v$  to *EXIT* starting with edge  $v \rightarrow w_k$  will generate a unique value in the range  $\sum_{i=1}^{k-1} NumPaths(w_i) \dots (\sum_{i=1}^k NumPaths(w_i)) - 1$ . Since all  $NumPaths(w_i)$  values are greater than 0, it follows that no two paths from  $v$  to *EXIT* generate the same value sum.  $\square$

Figure 6 illustrates how the algorithm operates on the example control-flow graph. Note that vertices are labeled in topological ordering, so  $FEDCBA$  is a reverse topological order. Any vertex with a single outgoing edge  $e$ , such as  $C$  and  $E$ , always has  $Val(e) = 0$ .

### 3.3 Efficiently Computing Sums

Given an edge value assignment, the second step of the algorithm finds a minimal cost set—with respect to a weighting (Section 3)—of edges along which to compute these values, while preserving the two properties of the value assignment.

This step of the algorithm finds a maximal cost spanning tree of the graph (to find a minimal cost set of chord edges), and applies an efficient event counting technique [Bal94] to determine the increment  $Inc(c)$  for each chord  $c$  in a spanning tree. The event counting algorithm ensures that the sum of  $Inc$  values for any path  $P$  from *ENTRY* to *EXIT* is identical to the sum of  $Val$  values for  $P$ . Note that some of the  $Inc$  values may be negative, as in Figure 4. The edge  $EXIT \rightarrow ENTRY$  is required for this step (if this edge is

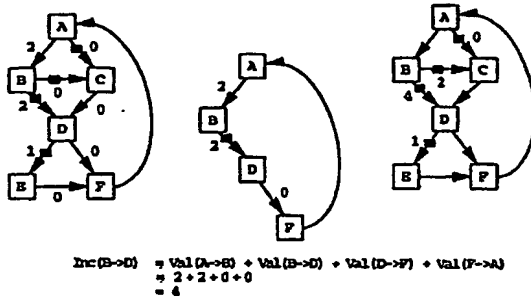


Figure 7. Application of event counting algorithm to determine chord increments.

selected as a chord, then its instrumentation can be placed in the *EXIT* vertex).

Figure 7 shows how the event counting algorithm applies to the example control-flow graph. The graph on the left contains the value assignment and the chord edges (those with squares along them). The graph in the middle shows the unique cycle of spanning tree edges associated with chord  $B \rightarrow D$ . The values along edges in this cycle determine the increment for chord  $B \rightarrow D$ , which in this case is four. Informally, the algorithm propagates the value of two from edge  $A \rightarrow B$  to the chord  $B \rightarrow D$ . The graph on the right contains increments for all chords. Each path from  $A$  to  $F$  in this graph yields the same path sum as in the graph on the left.

### 3.4 Instrumentation

After computing chord increments, the algorithm selects instrumentation. Of course, at the start of a program's execution, the array of counters must be allocated and initialized to 0. At program termination, this array is written to permanent storage.

Besides this prelude and postlude instrumentation, the remaining instrumentation has three tasks: initializing path register  $r$  [ $r = 0$ ] in the *ENTRY* vertex; updating  $r$  in chord  $c$  [ $r += \text{Inc}(c)$ ]; and incrementing a path's memory counter in the *EXIT* vertex [ $\text{count}(r)++$ ]. However, in many cases, an optimization can combine updates with the other two operations, as shown in Figure 2.

The optimization for initialization is:

- (1) A chord  $c$  may initialize the path register [ $r = \text{Inc}(c)$ ] if and only if  $c$  is the first chord in every path from *ENTRY* to *EXIT* containing  $c$ .

If there exists a path in which chord  $c$  is the first chord and another path in which  $c$  is not the first chord, then chord  $c$

must update rather than initialize  $r$ . However, in this case, moving the initialization [ $r = 0$ ] as close to  $c$  as possible avoids redundant initialization of  $r$ .

A similar optimization works for counter increments:

- (2) A chord  $c$  may increment the path register and memory counter [ $\text{count}(r + \text{Inc}(c))++$ ] if and only if  $c$  is the last chord in every path from *ENTRY* to *EXIT* containing  $c$ .

This optimization can fold an addition into a memory address calculation.

If a chord  $c$  contains initialization as well as a counter increment, the instrumentation simply becomes  $\text{count}[\text{Inc}(c)]++$ .

The algorithm in Figure 8 places instrumentation properly. The first while loop moves initialization code to chord edges when possible, and otherwise moves it far enough from the *ENTRY* vertex so that no initialization is redundant. The second loop places the memory increment code. The invariant of the first loop is that for each vertex  $w$  added to the working set, there is only one path from *ENTRY* to  $w$  in the DAG and this path contains no chords. Note that if there are two paths from *ENTRY* to a vertex  $w$ , one of these paths must contain a chord, so any chord encountered from  $w$  onward cannot satisfy this condition. A similar invariant is maintained by the second while loop.

Figure 9 shows the instrumentation for the control-flow graph from Figure 1. Note that edges  $A \rightarrow C$ ,  $B \rightarrow C$  and  $B \rightarrow D$  now initialize register  $r$ , eliminating the need to initialize  $r$  at vertex  $A$ . Furthermore, the update along edge  $D \rightarrow E$  has been combined with the counter increment code. However, notice that a counter increment is required along edge  $D \rightarrow E$  as well.

### 3.5 Regenerating a Path

To recreate a path profile from the path counters recorded at run time, it is necessary to map from the integer representing a path to the path itself. This is done using the value assignment computed previously (Section 3.2).

The regeneration algorithm is straight forward. Regeneration starts from a control flow graph's *ENTRY* node and traverses the graph, using the path value to select which edge to follow out of a basic block. Let  $v$  be a vertex in the reconstructed path and let  $R$  be the path value. Initially,  $v = \text{ENTRY}$  and  $R$  is the number of the path to regenerate. At each block, find  $e = v \rightarrow w$ , which is the outgoing edge of  $v$  with the largest  $\text{Val}(e) \leq R$ . As the path traverses edge  $e$ , let  $v = w$  and  $R = R - \text{Val}(e)$ . Repeat this process until control reaches the *EXIT* vertex.

For example, consider the control-flow graph in Figure 6. Suppose that the initial path value  $R$  is 3. At vertex  $A$ , the algorithm will choose edge  $A \rightarrow B$  and decrement  $R$  by 2.

```

// Register initialization code
//
WS.add(ENTRY);
while not WS.empty() {
  vertex v = WS.remove();
  for each edge e = v->w
    if e is a chord edge
      instrument(e, 'r=Inc(e)');
    else if e is the only incoming edge of w
      WS.add(w);
    else instrument(e, 'r=0');
}

// Memory increment code
//
WS.add(EXIT)
while not WS.empty() {
  vertex w = WS.remove();
  for each edge e = v->w
    if e is a chord edge {
      if e's instrumentation is 'r=Inc(e)'
        instrument(e, 'count[Inc(e)]++');
      else
        instrument(e, 'count[r+Inc(e)]++');
    } else if e is the only outgoing edge of v
      WS.add(v);
    else instrument(e, 'count[r]++');
}

// Register increment code
//
for all uninstrumented chords c
  instrument(c, 'r+=Inc(c)')

```

Figure 8. Algorithm for placing instrumentation.

At vertex  $B$ ,  $R = 1$ , so the algorithm traverses edge  $B \rightarrow C$  and then  $C \rightarrow D$ . At vertex  $D$ ,  $R$  still has a value of 1; so the path traverses edge  $D \rightarrow E$ , followed by  $E \rightarrow F$ . The resulting regenerated path is  $ABCDEF$ , which is the path that generates the path sum 3.

### 3.6 Early Termination

Like other efficient profiling algorithms [BL94], path profiling requires extra information to derive correct profiles for routines that terminate unexpectedly because of exceptions, unrecognized non-local gotos, or calls to exit. This information consists of the address of unterminated calls and can easily be obtained from a program's stack at an unexpected event. The event counting algorithm provides a way to correctly update the counters in these routines [Bal94].

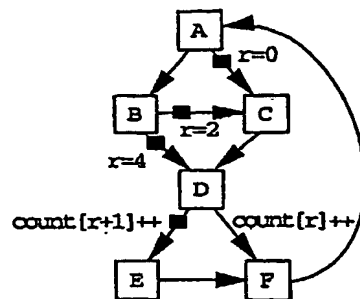


Figure 9. Optimization of instrumentation for the control-flow graph of Figure 1.

## 4 Path Profiling of Arbitrary Control-Flow

This section extends path profiling to arbitrary control-flow graphs that contain cycles (including irreducible loops). Any cycle in a control-flow graph must contain a backedge (as identified by a depth-first search of the graph). The algorithm in Section 3 only works for acyclic paths, which correspond to backedge-free paths.

Our approach to handling general CFGs instruments each backedge with a path counter increment and path register initialization [ $\text{count}[r]++$ ;  $r = 0$ ], which records the path up to the backedge and prepares to record the path after the backedge.

Suppose that  $v \rightarrow w$  and  $x \rightarrow y$  are backedges. A general CFG contains four possible types of acyclic (backedge-free) paths:

- A path from *ENTRY* to *EXIT*.
- A path from *ENTRY* to  $v$ , ending with execution of backedge  $v \rightarrow w$ .
- A path from  $w$  to  $x$  (after execution of backedge  $v \rightarrow w$ ), ending with execution of backedge  $x \rightarrow y$  (note:  $v \rightarrow w$  and  $x \rightarrow y$  may be the same edge).
- After executing backedge  $v \rightarrow w$ , a path from  $w$  to *EXIT*.

Removing all backedges from a control-flow graph produces a DAG (as defined in Section 3.1). However, simply applying the profiling algorithm from Section 3 to this DAG will not correctly distinguish the above four types of paths. Figure 10(a) contains a control-flow graph with a loop consisting of the vertices  $B$ ,  $C$ ,  $D$ , and  $E$ . Suppose the graph is instrumented by eliminating the backedge  $E \rightarrow B$ , thus yielding a DAG, and applying the path profiling algorithm for DAGs. The resulting assignment does not ensure that different paths yield different paths sums. For example, the

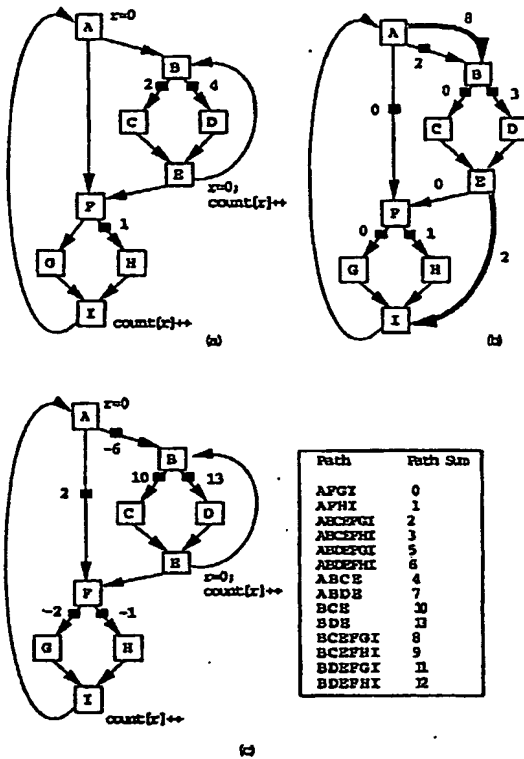


Figure 10. Control-flow graph with a loop.

paths *BCE*, *ABCE*, *BCEFGI* and *ABCEPHI* all compute the identical path sum of 2. Not surprisingly, only paths that start at *ENTRY* = *A* and end at *EXIT* = *I* are correctly distinguished.

The solution to this problem consists of three steps:

- For each vertex *v* that is the target of one or more backedges, add a dummy edge *ENTRY* → *v*. For each vertex *w* that is the source of one (or more) backedges, add a dummy edge *w* → *EXIT*. If one of these edges is not in the spanning tree, it will be instrumented, which is efficient as the edge's increment can be combined with the code always added to a loop backedge to record a path.
- Eliminate backedges from the graph (except for the edge *EXIT* → *ENTRY*, which was added for increment computation).
- Apply the first two steps of the path profiling algorithm (Section 3) to compute a value assignment and chord increments.

The dummy edges create extra paths from *ENTRY* to *EXIT*, which the value assignment algorithm takes into account. The dummy edge from *ENTRY* to a loop head corresponds to reinitializing the path register along the loop's backedge. The dummy edge from the loop's bottom to *EXIT* corresponds to incrementing the path counter along the backedge.

Figure 10(b) shows the graph after this transformation and edge value assignment. Dummy edges are the thicker edges. As a result, the chord increments correctly distinguish the four classes of paths listed above. Figure 10(c), shows the chord increments computed and the path sum for each possible path through the graph.

Path regeneration must follow the first two steps (adding dummy edges and removing backedges) to compute the same value assignment, before using the regeneration algorithm from Section 3.5 on the resulting graph.

#### 4.1 Self Loops

The approach described must be slightly modified to handle self-loop edges, which are backedges with the same source and target vertex. Removing this edge does not leave any edge in the loop to instrument. These edges can be handled specially, by adding a counter along them to record the number of times they execute, rather than instrumenting them with the code `[count[x]++; x = 0]`.

### 5 Implementation

We implemented the algorithms described previously in a tool called PP, which instruments SPARC binary executables to extract path profiles. PP is built on EEL (Executable Editing Library), which is a C++ library that hides much of the complexity and system-specific detail of editing executables [LS95]. EEL provides abstractions that allow a tool to analyze and modify binary executables without being concerned with particular instruction sets, executable file formats, or the consequences of deleting existing code and adding foreign code (i.e., instrumentation).

#### 5.1 Registers

Path profiling requires a local register throughout each routine's execution to hold the current path and a temporary register for some instrumentation code, such as the memory increment code. EEL scavenges free registers by using dataflow analysis to find the dead registers throughout a control-flow graph. If EEL cannot find an unused local register, it frees the least heavily used local register by spilling it to the routine's stack frame. In most routines, EEL found unused local registers, although many larger and computationally intensive routines require spill code. The SPARC's reg-

ister windows ensure that all local registers are caller saved. Other architectures would need to save the path register before and after calls.

EEL also provides a facility to add procedure calls at arbitrary points in a routine. PP uses this feature, which relies on program analysis to save only live values, to call the hashing code (see Section 5.3).

## 5.2 Optimizations

A simple strength-reduction optimization saves two instructions per path by having the path register hold a counter's address, instead of its index. PP initializes a path register to the base of the counter array. Each increment adds its update, scaled by the size of a counter (4 bytes). This optimization saves three instructions in the code that increments a path's counter, at the cost of an additional instruction in the code that initializes the path register. Unfortunately, the optimization reduces the range of increments that fit in an instruction's immediate field. Since the SPARC's immediate field is 13 bits, this optimization is limited to routines in which the largest increment is 1023 (rather than 4095). However, most routines have fewer or far more paths.

Moreover, a simple change can reduce the range of increments in a routine. The algorithm in Figure 5 can visit a node's successors in any order. By visiting the successor with the largest number of paths (`NumPath`) last, the value (`Val`) assigned to the last edge is minimized, and hence so are the increments added at run time.

## 5.3 Routines with Many Paths

PP employs two techniques to handle routines with a large number of paths. The first, which PP applies to any routine in which an increment is larger than an instruction's immediate field, replaces the array of counters with a hash table. On the SPARC, this means that routines with more than 4000–6000 paths require hash tables. Hash tables have the advantage of requiring space proportional to the number of executed paths, but have the disadvantage of being an order of magnitude more costly than a simple memory increment and forcing a function call in awkward places. This technique limits counter space for a routine to roughly 16K bytes (most routines require far less space). PP uses two hash routines. The one called from loop backedges keeps a pointer to the last path and hash bucket, so that repeated lookups of the same path are very fast. This is very beneficial since procedures often spend their time in tight loops, repeating the same path over and over.

PP's other technique is necessary for very complex routines in which the number of possible paths exceeded the range of a 32 bit integer. In these routines, PP terminates

the value computation (Section 3.2) when the number of paths reachable from a node exceeded a threshold (currently 100,000,000). At this point, PP removes all outgoing edges from the node—using the same approach to terminate these paths as for loop backedges (Section 4)—and reruns the value computation. The only information lost was the relation of the path before a cut edge with the path after the cut edge. Larger (64 bit) words would alleviate, though probably not eliminate, the need to truncate paths.

## 6 Experimental Results

This section uses the SPEC95 benchmarks to compare path profiling (PP) against edge profiling (QPT2), which has the lowest overhead of conventional profiling techniques [BL94]. The programs ran stand-alone on a Sun Ultra-server E5000–167Mhz UltraSPARC processors and 2GB of memory—running Solaris 2.5.1 with a local file system. Table 1 presents measurements of the SPEC95 benchmarks using the *ref* input data.<sup>2</sup> C benchmarks were compiled with gcc (version 2.7.1) and Fortran benchmarks were compiled with Sun's f77 (version 3.0.1). Both compilers used only the -O option.

PP's overhead (across all input files) averaged 30.9% (5.5–96.9%) and QPT2's overhead averaged 16.1% (-2.6–52.8%). PP's overhead averaged 2.8 times QPT2's overhead (0.7–14.5). PP's overhead is explained in part by the final two columns in Table 1, which report the fraction of path increments that required hashing and the average number of instructions between increments. However, the table does not report the cache interference caused by profiling code and data. In general, programs with little hashing (e.g., compress, li, jpeg, turb3d) have PP overhead comparable or lower than QPT2. Programs with considerable hashing (e.g., tomcatv, fpppp, and wave5) can still have low overheads if blocks are large or paths are long and path increments execute infrequently.

Table 2 reports some characteristics of the program's acyclic paths. In all programs, the number of executed paths was small (fewer than 2300 in all except 099.go and 126.gcc) and is dwarfed by the potential paths—which number hundreds of millions to tens of billions, even after path truncation.

The table also reports the length of the longest and average acyclic paths. Not surprisingly, the weighted number of instructions in a path in the CFP95 benchmarks, 91.7 (43.1–636.0), is significantly longer than CINT95 benchmarks, 21.4 (15.1–33.2). More aggressive compiler optimizations would further increase these numbers by loop unrolling and procedure in-lining.

<sup>2</sup> Since PP measures a single process's execution, the tables report program behavior for each benchmark's last input file.

Benchmark	Base Time (sec)	PP Overhead %	QPT2 Overhead %	PP/QPT	Path Inc (million)	Edge Inc (x Path)	Hashed Inc %	Inst/Inc
099.go	885.0	53.4	24.1	2.2	1002.4	1.5	27.7	33.2
124.m88ksim	571.0	35.6	18.7	1.9	4824.9	1.2	3.9	16.2
126.gcc	322.0	96.9	52.8	1.8	9.4	1.7	16.8	15.1
129.compress	351.0	19.4	21.9	0.9	3015.7	1.5	0.0	16.6
130.li	480.0	25.4	26.7	1.0	3282.4	1.4	1.2	16.8
132.jpeg	749.0	17.4	16.3	1.1	1164.9	1.1	1.2	31.0
134.perl	332.0	72.9	51.5	1.4	1133.0	1.9	23.4	22.2
147.vortex	684.0	37.7	34.1	1.1	3576.3	1.5	23.7	20.3
CINT95 Avg:		44.8	30.8	1.4	22251.1	1.5	12.2	21.4
101.tomcatv	503.0	19.9	2.8	7.1	574.6	1.1	95.8	93.0
102.swim	691.0	8.4	0.6	14.5	163.4	1.0	0.2	162.9
103.su2cor	465.0	10.1	5.8	1.7	558.1	1.2	21.5	92.8
104.hydro2d	811.0	37.7	5.8	6.5	1690.7	1.7	77.8	43.1
107.mgrid	872.0	6.3	3.2	2.0	1035.2	1.0	7.7	133.5
110.applu	715.0	71.0	12.0	5.9	2111.4	1.1	99.1	44.8
125.turb3d	1066.0	5.5	7.4	0.7	2952.8	1.1	0.0	56.5
141.apsi	492.0	7.7	1.8	4.2	599.3	1.1	3.5	84.0
145.fpppp	1927.0	14.6	-2.6	-5.6	395.0	1.8	42.5	636.0
146.wave5	620.0	16.9	6.1	2.8	737.3	1.3	65.0	74.1
CFP95 Avg:		19.8	4.3	4.0	1081.8	1.2	41.3	142.1
Average:		30.9	16.1	2.8	1601.5	1.3	28.4	88.4

**Table 1.** Comparison of path profiling (PP) against Ball-Larus edge profiling (QPT2). *Base Time* is elapsed time of uninstrumented program with the *ref* dataset. *Overhead* is the increase in execution time due to profiling. *PP/QPT2* is the ratio of the overheads. The remaining numbers report a program's behavior on its last input file. *Path Inc* is the number of increments of path counters. *Edge Inc* is the ratio of edge profiling increments to path increments. *Hash Inc* is the fraction of path increments that required hashing. *Inst/Inc* is the average number of instructions between increments.

Benchmark	Num Path	Path Profile				% Correct	Edge Profile Paths				Routines		
		Longest	Avg	Longest	Avg		Longest	Avg	Longest	Avg	Exec	Max Path	Avg Path
099.go	24414	105	314	10.9	33.2	4.3	84	252	5.0	19.3	407	1574	60.0
124.m88ksim	1113	138	360	5.8	16.2	29.8	138	360	4.3	9.1	220	70	5.1
126.gcc	9319	711	1074	7.4	15.1	20.8	711	1074	4.6	10.5	1027	163	9.1
129.compress	249	80	146	6.5	16.7	43.0	80	146	4.6	8.8	69	34	3.6
130.li	770	153	252	9.0	16.8	38.1	62	109	7.0	14.6	216	64	3.6
132.jpeg	1199	139	416	7.0	31.0	36.4	139	202	5.1	22.2	252	194	4.8
134.perl	1421	123	305	10.7	22.2	24.5	115	207	7.3	16.7	233	125	6.1
147.vortex	2223	584	841	8.9	20.3	39.5	584	841	8.5	15.6	627	65	3.5
CINT Avg:	5088	254	464	8.3	21.4	29.5	239	399	5.8	14.4	381	286	12.0
101.tomcatv	421	83	326	4.1	93.0	49.6	82	201	3.9	25.2	146	56	2.9
102.swim	378	106	310	2.3	162.9	57.1	106	310	2.2	57.6	143	25	2.6
103.su2cor	905	136	954	6.7	92.8	45.9	73	781	5.5	59.3	209	127	4.3
104.hydro2d	1456	344	488	6.5	43.1	33.2	344	436	5.8	36.2	227	434	6.4
107.mgrid	589	83	320	2.3	133.5	44.8	78	320	2.1	15.8	160	73	3.7
110.applu	619	240	3557	3.7	44.8	54.1	240	3557	2.4	26.6	144	82	4.3
125.turb3d	674	162	692	7.1	56.5	46.6	162	692	5.1	28.2	189	39	3.6
141.apsi	1064	712	1196	6.1	84.0	40.8	136	734	4.6	65.0	242	54	4.4
145.fpppp	821	85	11455	14.9	636.0	25.8	76	11455	9.0	121.6	143	322	5.7
146.wave5	896	90	1180	5.2	74.1	47.8	90	1180	4.8	45.5	212	69	4.2
CFP Avg:	782	204	2048	5.9	142.1	44.6	139	1967	4.5	45.0	182	128	4.2
Average:	2696	226	1344	6.9	88.4	37.9	183	1270	5.1	32.6	270	198	7.7

**Table 2.** Characteristics of executed acyclic paths. Paths are from the last input file in the *ref* dataset. *Num Paths* is the number of paths executed. *Longest* is the longest executed path. *Avg* is the average path length, weighted by execution frequency. *% Correct* is the fraction of paths predicted entirely correctly by edge profiling. Statistics on edge-predicted paths are up to the first misprediction. The final columns report the executed routines, and maximum and average number of paths per executed routine.

Benchmark	Common Paths			Common Instructions	
	Number	Static	Dynamic	Number	Dynamic
099.go	13670	52.8%	99.5%	32823357655	98.6%
124.m88ksim	800	71.1%	97.1%	68794935104	93.3%
126.gcc	9058	63.5%	94.6%	140747413	88.9%
129.compress	201	78.2%	99.2%	49206874856	98.0%
130.li	522	67.8%	87.4%	48285966561	87.7%
132.jpeg	1099	85.3%	99.9%	36098785257	100.0%
134.perl	687	41.8%	71.9%	16041579783	63.8%
147.vortex	2160	96.0%	100.0%	72589443060	100.0%
101.tomcatv	418	90.9%	100.0%	53422202701	100.0%
102.swim	373	98.2%	100.0%	26607743709	100.0%
103.su2cor	880	95.5%	100.0%	51766673091	100.0%
104.hydro2d	1336	90.1%	100.0%	72892099182	100.0%
107.mgrid	577	97.3%	100.0%	138157253535	100.0%
110.apple	582	87.5%	100.0%	94622106074	100.0%
125.turb3d	651	94.5%	100.0%	166724577396	100.0%
141.apsi	992	91.0%	91.2%	37648860455	74.3%
145.fpppp	764	91.9%	99.9%	250229811735	99.6%
146.wave	5821	88.5%	98.8%	52996842799	96.9%

**Table 3.** Comparison of paths with the SPEC95 *ref* and *train* input datasets. *Common Paths* reports the number of paths executed for both input datasets. *Static* is the fraction of executed *ref* dataset paths executed by the *train* dataset. *Dynamic* is this fraction, weighted by execution frequency. *Common Instructions* reports the number and fraction of instructions along common paths.

The table also compares paths predicted using edge profiling against measured paths. For this experiment, PP followed each executed path, starting at a function entry or loop head, and used the most frequently executed edge out of a block to predict the next step in a path. If two edges had the same frequency, PP followed the not taken edge. This approach predicts, in their entirety, an average of 37.9% (4.3–57.1%) of the paths. PP also computed the length of a predicted path, up to their first mispredicted edge. Weighted by execution frequency, edge predicted paths were nearly as long as measured paths (5.1 versus 6.9 CFG edges), but contained significantly fewer (33.6 versus 88.4) instructions. In part, this result reflects the simple behavior of the SPEC95 benchmarks, which execute an average of 7.7 (2.6–60.0) paths in each routine. It is also consistent with previous work in dynamic and static branch prediction [FF92, BL93], which found that branches in a program typically follow one direction with very high probability, and this high-probability direction generally remains the same for different inputs.

Table 3 compares paths executed with the SPEC95 *train* and *ref* input datasets. In the CINT benchmarks, the *train* dataset identified 41.8–96.0% of the paths executed in the *ref* dataset. However, these common paths were the important ones that account for 71.9–100.0% of path executions and 63.8–100.0% of instruction executions. The CFP benchmarks were even more striking, as the *train* dataset identified a minimum of 87.5% of the executed paths, which in all except one case (147.apsi) account for 99+% of instructions. This result is again consistent with earlier work

that found program behavior to be independent of program data [FF92, BL93]. These measurements suggest that path profiles could greatly improve peak SPEC95 performance by providing an inexpensive and accurate basis for profile-driven compilation.

## 7 Summary

This paper describes a new algorithm for path profiling. A path profile records the execution frequencies of acyclic paths in a routine. Although these profiles provide far more information than basic block or edge profiles, they can be obtained at a cost comparable to the best known profiling algorithms.

We also compared path profiles against paths predicted by a common heuristic that selects the highest frequency edge out of a basic block. Path profiles yielded slightly longer paths that contained significantly more instructions. If edge profiling was significantly cheaper or easier than path profiling, it might be a reasonable heuristic. However, since path profiling is similar in cost and complexity to edge profiling, there is little reason to forgo the benefits of measurement.

Moreover, path profiles obtained from short runs on short, training datasets covered most of the paths and instructions found in full runs. The profiles from these short runs form an inexpensive and accurate basis for profile-driven compilation.

## Acknowledgements

Many thanks to Vasanth Bala for helpful discussions about path profiling. Chris Lukas and David Wood provided helpful comments on this paper.

## References

- [Bal94] Thomas Ball. Efficiently counting program events with support for on-line queries. *ACM Transactions on Programming Languages and Systems*, 16(5):1399–1410, September 1994.
- [Bal96] Vasanth Bala. Low overhead path profiling. Technical report, Hewlett Packard Labs, 1996.
- [BL93] Thomas Ball and James R. Larus. Branch prediction for free. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, pages 300–313, June 1993.
- [BL94] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.
- [Cha88] Pohua P. Chang. Trace selection for compiling large C application programs to microcode. In *21th Annual Workshop on Microprogramming and Microarchitecture (MICRO 21)*, pages 21–29, November 1988.
- [Ell85] John R. Ellis. Bulldog: A compiler for VLIW architectures. Technical Report YALEU/DCS/RR-364, Yale University, Department of Computer Science, February 1985.
- [FF92] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 85–95, October 1992.
- [Fis81] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.
- [Gol91] Aaron Goldberg. Reducing overhead in counter-based execution profiling. Technical Report CSL-TR-91-495, Computer System Laboratory, Stanford University, October 1991.
- [LS95] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.
- [YS94] Cliff Young and Michael D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 232–241, October 1994.

# Speeding Up Program Execution Using Reconfigurable Hardware and a Hardware Function Library

Sitanshu Jain

M. Balakrishnan  
Shashi Kumar

Anshul Kumar

Department of Computer Science and Engineering  
Indian Institute of Technology  
New Delhi 110016  
email: mbala@cse.iitd.ernet.in

## Abstract

This paper describes a CoDesign environment which follows a new approach for speeding up compute intensive applications. The environment consists of three major components. First, a target architecture consisting of a uniprocessor host and a board with dynamically reconfigurable FPGAs and memory modules; second, a library of functions pre-synthesized for hardware or software implementation; and third, a tool which takes as input an application described in C and partitions it into hardware and software parts at functional granularity using information obtained by profiling the application. An important feature of the partitioning tool is a new efficient heuristic specifically suited for the architecture with reconfigurable hardware.

## 1 Introduction

Real time applications with strict performance constraints usually calls for CoDesign in order to achieve the right tradeoff between faster and expensive special purpose hardware and cheaper and flexible software. The main task in CoDesign revolves around hardware-software partitioning. A common technique used for partitioning is to extract the computationally expensive portions of the application into hardware, with the remaining executing in software, while meeting all constraints [2]. Alternatively, there are approaches which start from VHDL [3] or Hardware C [1]. Another feature which distinguishes the various CoDesign systems is the partitioning granularity, which varies from fine grain [2] to coarse grain [4, 1]. In the target architecture<sup>1</sup>, usually the processor(s) is predecided, whereas the hardware is customized for the application [2].

<sup>1</sup>also referred to as templates in this paper

In our approach, a set of functions are pre-synthesized into hardware designs and organized as a library of hardware functions (called Hardware Function Library or HLIB). This considerably simplifies the overall development cycle. Further, it makes it possible to use exact cost/performance measures during partitioning rather than estimates. This also implies that the partitioning granularity is at function level. We exploit the reconfigurability of the FPGAs to reuse the hardware by dynamically loading different functions at different times during the execution of the application.

In the next section, we give an overview of our complete system, describing the target architecture, the hardware function library, and organization of the software. The algorithm behind this tool is discussed in Section 3. In Section 4, some implementation issues are discussed, and a complete example is shown in Section 5. We conclude in Section 6 with scope for future work on the system.

## 2 System Overview

### 2.1 Target Architecture

The target architecture assumed in this paper consists of a single processor with a set of FPGAs interfaced through the system bus [?]. Each FPGA is assumed to have a local memory used for all the data exchanges between the processor and FPGA. This memory is mapped to the processor address space also. A controller, again implemented with an FPGA, coordinates access to the bus, interprets commands for loading FPGAs and monitors status of computations on FPGAs. The architecture assumed is shown in Figure 1. In case a task implemented on an FPGA is to be initiated, the processor fills the FPGA local memory with the data arguments, transfers control to the

FPGA, and waits till it returns. At the time of configuration, the processor copies configuration files to the local memory of the FPGAs and gives a configuration command to load these files onto the FPGAs.

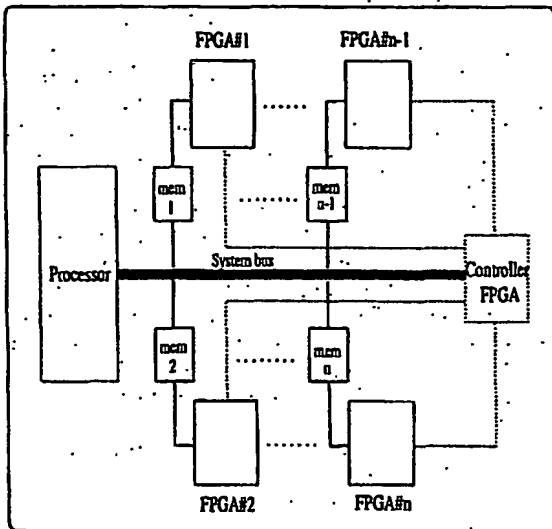


Figure 1: Target Architecture

## 2.2 Hardware Function Library (HLIB)

The Hardware Library is created by identifying functions which are computation intensive; hardware implementable and commonly used in a given application area. Each function is pre-synthesized for a specific FPGA type. For each function, the following information is stored in HLIB :

- Configuration file.
- Compiled code for the processor.
- Performance parameters such as software execution time and hardware execution time.
- Memory address map and size of the parameters.
- Information regarding the function's behavior pertaining to each parameter.

## 2.3 Software Organization

We use C as the input specification language, with some restrictions on the style of modeling. Structures and user defined types are not supported. Typecasting too is not allowed. The user is also expected to follow a specific modeling scheme, which makes implementation easier, but does not limit the power of the language in any way.

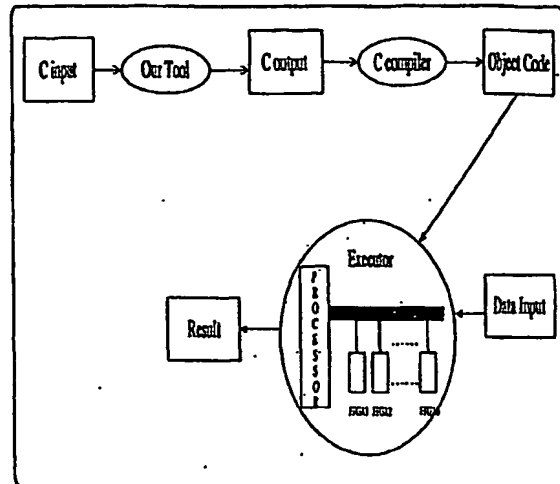


Figure 2: Design flow

The design flow of this approach is described in Figure 2 which also indicates the function performed by our tool. In Figure 3, the various modules comprising the tool are depicted.

We have divided the tool into five major modules, namely :

- **Profiler** : generates statistics about the execution behavior of the input application, e.g., the average number of times a loop is executed and the number of times a conditional is successful.
- **Parser** : generates a *syntax tree* (AST) which represents the input in a dynamic tree structure. It also generates a *list of functions* used as an input to the partitioner module, and a *symbol table*.
- **Partitioner** : assigns hardware or software implementation to library functions in the input, based on a heuristics to improve overall performance. HLIB provides quantitative information for partitioning, such as time taken by software and hardware implementations etc.
- **Tree Transformer** : takes the result of partitioning and modifies the AST to ensure the desired implementation of each HLIB function. Manipulations need to be performed to handle data allocation and consistency, since the processor can access the local memory of the FPGAs, but an FPGA has access to only its local memory.
- **Code Generator** : writes the transformed AST

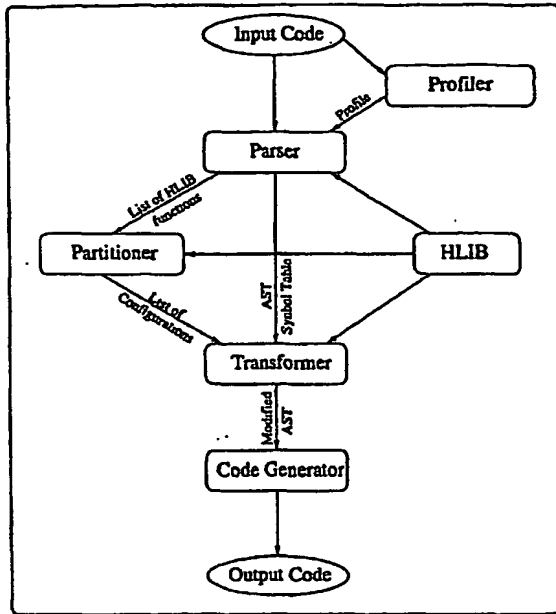


Figure 3: System Schematic of our tool

to a C file which is compiled to get the code on the host processor.

### 3 The Partitioning Algorithm

The algorithm tries to overcome the drawbacks of both exponential time algorithms and greedy ones. It uses sufficient lookahead to avoid local minimas, but iterates only once to obtain fast solutions. Fast algorithm acts as an aid to user in the decision process. The algorithm is divided into two phases : in the first phase, the input is parsed to generate some data structures that aid in lookahead. In the second phase, these are used to decide on the implementation of each HLIB function. The software and hardware execution time is available in HLIB. Also available is the function dependent configuration time for each hardware implementation. Moreover, there is a common configuration time overhead for all functions, independent of data.

Two possibilities arise when we consider partitioning : we allow only static one time configuration, or can allow reconfiguration to be performed during the execution of the application. This choice arises because reconfiguration takes time, and is not useful unless the functions reconfigured are used extensively until the next reconfiguration. Also as we will discuss later, reconfiguration can lead to complications in control dominated applications, and much time may be

wasted in order to maintain consistency. On the other hand, static configuration has the drawback of poor utilization of hardware resources.

#### 3.1 Static Configuration

When only static (one time) configuration is used, the overhead of configuration time becomes a one time overhead, and so its offset can be added to all functions in the code. From the statistics obtainable from HLIB, and the profile information which gives the total number of calls to each function in the input, it is a simple job to obtain the gain for each function that will be generated if that function is to be mapped onto hardware. From each of these gains, the time to configure that function is subtracted to give an estimate of the effectiveness of the hardware mapping of the function. Now for a target hardware with  $n$  FPGAs, at most  $n$  most effective functions can be picked and configured on the FPGA board at the beginning of the code. Multiple functions implemented on the same FPGA is not allowed since it leads to design contentions in terms of the pins and area available.

#### 3.2 Statically Determined Runtime Reconfiguration (SDRR)

In this approach we allow reconfiguration to be performed at runtime but the decision is taken statically, that is at translation time. Reconfiguration means that the same FPGA may be used for multiple functions during the lifetime of the program, with a configuration file loaded with each new function. Since reconfiguration involves loading of configuration file onto the FPGA local memory, it may not be advisable to reconfigure unless the hardware implementations of the functions involved score greatly over their software implementations. The partitioning problem is therefore two fold - to decide on function implementation, and to decide on the positioning of reconfigurations. We developed a heuristics that does both in polynomial time. Before we describe the heuristics, we define the following :

- **Configuration** : A set of two array variables namely FPGA and CHANGED, which are indexed over number of FPGAs.  $FPGA[i]$  gives the function implemented in FPGA # $i$  at that moment.  $CHANGED[i]$  is a flag that gets set when the contents of  $FPGA[i]$  are either changed or reused.
- **Incoming Weight** : weight attached to a function being processed for implementation determination (in units of time gain that will be produced if this function is implemented on FPGAs.)  
 $W_{in} = W_{cur\_gain} + W_{fut\_gain} + W_{dist} - C_{conf}$   
 where :  
 $W_{cur\_gain}$  : gain by implementing the function in

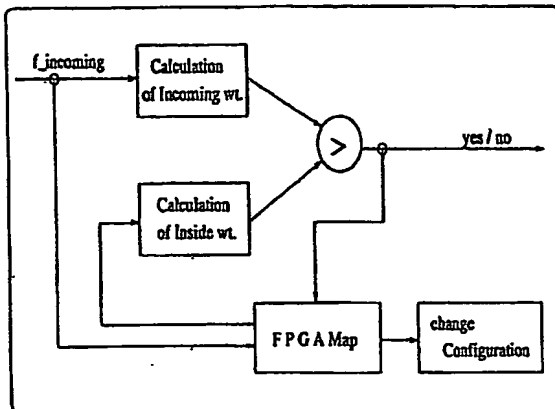


Figure 4: Calculator for sequences of functions

hardware.

$W_{fut\_gain}$  : gain due to later calls to this function.

$W_{dist}$  : weight corresponding to the distance of next call of the function.

$C_{conf}$  : Configuration cost.

$C_{conf} = k * \text{common configuration time} + \text{data configuration time}$ , where  $k$  is an empirical constant,  $0 \leq k \leq 1$ .

- **Inside Weight** : weight attached to a function already implemented on an FPGA.

$$W_{inside} = W_{fut\_gain} + W_{dist}$$

- **Configuration Weight** : weight of a configuration. This can be defined in two ways based on the following policies :

**Reconfiguration Avoidance policy** : Reconfiguration is done only when  $CHANGED[i] = 1 \forall i$ .

$$W_{config} = \min\{W_{inside}(FPGA[i]) : CHANGED[i] = 0\};$$

**Reconfigure the weakest policy** : replace the function with the lowest weight, irrespective of whether it is being used in the current configuration or not.

$$W_{config} = \min\{W_{inside}(FPGA[i])\};$$

Now the heuristics can be described. There are three different control flow situations which may occur while traversing the list of functions. These are:

- A sequence of functions**

In this case we traverse the list of functions sequentially, calculating  $W_{in}$  and  $W_{config}$  using the *Reconfigure the weakest* policy (Figure 4).

- Conditionals**

In this case the strategy depends on whether the

*if path* is more frequent or the *else path* is more frequent or both are equally likely. At the merger of the two paths, the configuration retained is that of the *if path*, the *else path* or the intersection of the two configurations respectively.

- Loops**

Inside loops, reconfiguration is not desirable unless time spent inside the loop is large. Therefore we use both SDRR and Static configuration in the loop body, and choose the one with lower cost.

The time complexity of the above described algorithm is  $O(mn^2)$ , where  $n$  is the number of calls to the functions in HLIB, and  $m$  is the number of FPGAs. The analysis is omitted here for the sake of brevity.

## 4 Some Implementation Issues

In order to keep the tool portable, it is designed to produce a C output. This output incorporates the decisions generated by the partitioner through transformations at the AST level. These transformations involve additions, deletions and modifications of individual nodes, and are used to make the application execute faster on the template available.

Some major issues addressed in this phase are those of variable allocation, data copying and validity. The local memory of the FPGAs is mapped to the processor memory, and is accessible to it directly. Standard allocation routines are used for variables allocated in processor memory (not in FPGA local memory). Data validity is to be ensured in conditionals, since here the branch taken determines the currently valid location of data. We try to ensure this by keeping home locations where necessary, which keep a valid set of data at the end of conditionals.

Data copying needs to be minimized, because in many applications time taken for a two way data copy can far exceed the gains produced by implementing a function on FPGAs. A copy statement is to be added only when the function being considered is mapped to the FPGA, and the variable is a pointer (otherwise the data is anyhow being copied). Necessary conditions forcing a data segment copy into a location are:

- the data was never present in this location,
- the data was present in the location but it is not valid because its copy at some other location was updated after its last use in this location.

In all other cases, only pointer address need to be changed to the new location.

Example Input	HLIB functions
<pre> #include &lt;stdio.h&gt; #include "SLIB.h" main() {     int *a, *d;     int b, c, count;      a = (int *) malloc(100);     for (count = 0; count &lt; 10; count++) {         fx(a, b);         fy(c, a);         fz(d);     }     for (count = 0; count &lt; 50; count++) {         fx(a, c);         fz(d);     } } </pre>	<pre> HW_fx (int fpga_no, int *x, int y) /* changes x's data */ {     /* x's data is of size 100 bytes and starts from Xx H,     while y is to be copied to Xy H */      int *y_in_fpga = abs_addr(fpga_no, Xy); *y_in_fpga = y;      /* function body */ }  HW_fy (int fpga_no, int x, int *y) /* doesn't change x's data */ {     /* x is to be copied to Yx H, while y's data (size 100 bytes)     must start from Yy H */      int *x_in_fpga = abs_addr(fpga_no, Yx); *x_in_fpga = x;      /* function body */ }  HW_fz (int fpga_no, int *x) /* changes x's data */ {     /* x's data (size 2 bytes) starts from Zx H */      /* function body */ } </pre>

Figure 5: Example of an input program

## 5 An Illustrative Example

An example showing an input and the library assumed is shown in Figure 5 and the corresponding output generated by the system is depicted in Figure 6. For the sake of brevity, only a small example with a three function library and a two FPGA system is shown. Only the relevant statements are given, while the rest of the code remains unchanged by this tool. The library shows all the information required by the system. In table 1, statistics relating to HLIB functions is given. All values are in  $\mu s$ , and are conservative approximations of real examples used by us.

Function	$T_{software}$	$T_{hardware}$	$T_{data}$
<i>fx</i>	100	7	6.25
<i>fy</i>	159	16	6.25
<i>fz</i>	36	9.5	0

Table 1 : HLIB statistics

$T_{software}$  and  $T_{hardware}$  are the time required by the software and hardware implementations of the function respectively.  $T_{data}$  is the time required to setup data

arguments for the function. There is also a common configuration time required by the configure operation. This value is estimated to be around  $700\mu s^1$ .

It can be seen from the statistics that before the first loop, *fx* and *fy* are to be configured, and *fy* is to be replaced by *fz* before the second loop. Although hardware implementation of *fx* is much faster than software implementation, reconfiguration is still not performed inside the loop due to the large cost involved. Configuration is handled in the "Configure" function call. Also as was discussed in Section 5, malloc for 'a' is removed, and it is assigned the address in FPGA local memory where it is first required. The function "abs\_addr" gives the absolute address given an FPGA and a local address in it. All functions mapped to HLIB are linked to a different library (HLIB.h), and the *fpga\_no* is given as an argument for address calculation. Note that since argument 'a' is required in two distinct locations, it is copied. "Copy(a, b, n)" copies *n* bytes starting from

<sup>1</sup>Calculated using block transfer rates in the processor to copy configuration files to the local FPGA memory, and time taken to load them

# CONFIGURATION CODE GENERATION AND OPTIMIZATIONS FOR HETEROGENEOUS RECONFIGURABLE DSPS

Suet-Fei Li, Marlene Wan and Jan Rabaey

Berkeley Wireless Research Center and the Department of EECS  
University of California at Berkeley

**Abstract** - In this paper we describe a code generation and optimization process for reconfigurable architectures targeting digital signal processing and wireless communication applications. The ability to generate efficient and compact code is essential for the success of reconfigurable architectures. Otherwise, the overhead of reconfiguring could easily become the system bottleneck. Our code generation process includes the evaluation a set of tradeoffs in system design, software engineering as well as usage of a set of local and global optimization techniques. By doing so we are able to achieve results of significantly lower overhead.

## I. INTRODUCTION

Future networked embedded devices will need to support multiple standards of communication and digital signal processing. Reconfigurable systems have recently emerged as a promising implementation platform for such embedded computing by exhibiting both high performance [1] and low power [2][3] frequently required by such system-on-a-chip designs. The current trend of reconfigurable architectures in both general purpose computing and embedded digital signal processing is to combine a programmable processor with reconfigurable computing components of different granularities (fine-grain [4][5], datapath [6] and mixed [7]). The problem of interface generation between hardware and software has recently gained significant attention by the VLSI CAD community. The problem of integrating processors with reconfigurable elements has added another dimension to the interface generation problem - between software and configware. Careful configuration and interface code generation is essential [8] to ensure that the overhead of reconfiguring will not offset the speed and energy savings of reconfigurable components. This need is especially pronounced when the reconfiguration frequency is large within an application and when the timing constraints on the application are tight - which is often the case for real-time DSP and communication applications.

In this paper, we present a code generation and optimization process for reconfigurable architectures targeting digital signal processing applications. While the concept of our code generation and optimization process is machine-independent and can be applied to any reconfigurable architecture, we show the effectiveness of our code generation process on a heterogeneous reconfigurable

digital signal processor (Pleiades architecture template [9]). Similarly, while the process can be utilized as the backend to any system compilation tool, we use the software methodology proposed in [10] as the front end to obtain a good software-configurable partitioning. In the remainder of this paper, we will first introduce an overview of the architecture template and the software methodology in Section II. A detailed description of the proposed code generation and optimizations is given in Section III. At the end of this paper, we illustrate the effectiveness of this process by providing results for a baseband voice processing architecture.

## II. BACKGROUND

A heterogeneous reconfigurable digital signal processor template is shown in Figure 1. The template consists of a microprocessor and computational coprocessors of different programming granularities (referred to as satellites in the rest of this paper). The computation model on the processor is shown in Figure 2. A single thread can spawn computations on a cluster of satellite processors. The underlying satellite processors can support reconfigurations so multiple spawning within an application is possible (right side of Figure 2). From the programmer's point of view, reconfiguration means the configuration registers corresponding to each satellite (to specify the operation of the satellite) and the reconfigurable interconnect (to connect a cluster of satellites). Each one of the configuration registers is part of the microprocessor's memory map.

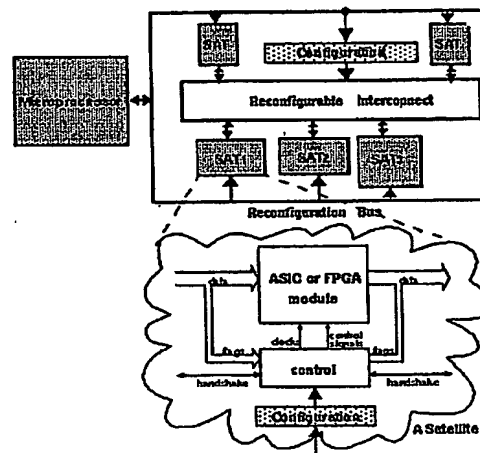


Figure 1 Heterogeneous Reconfigurable Architecture Template (Pleiades Architecture)

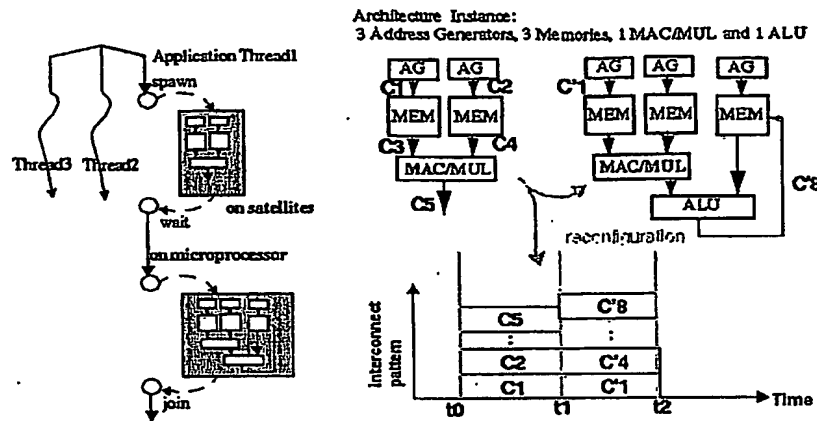


Figure 2 Model of Computation and Reconfiguration

Since the compilation of a single thread is the basis of other system-level scheduling tools that can utilize multiple threads, we concentrate our efforts in generating efficient interface code for a single application thread. For the above architecture template, the interface overhead mainly consists of the reconfiguration of satellites and the synchronization between the microprocessor and the satellites. In General, reduction of the overhead can be achieved by generating clever software or replacing parts of the software by special hardware dedicated to reconfiguration. In this paper, we focus on the software solution and discuss hardware enhancements in the future work section.

### III. CODE GENERATION AND OPTIMIZATIONS

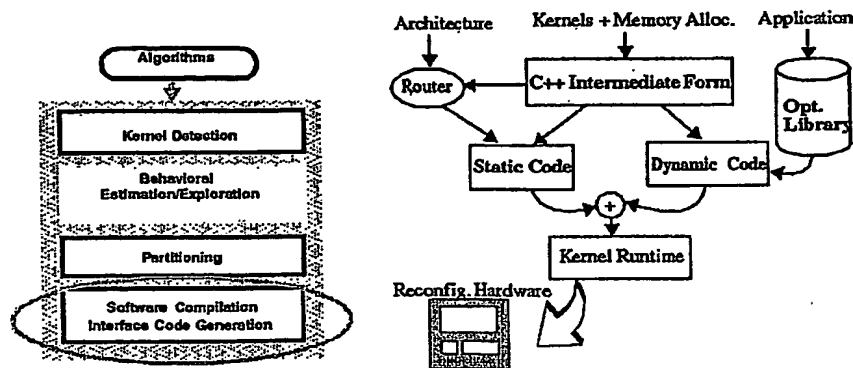


Figure 3 Software Methodology and the Backend Code Generation

## 1. Overall Software Methodology and the Intermediate Form

The overall software flow is shown on the left of Figure 3. The input to the software flow is an algorithm specified in C/C++. After mapping and partitioning across different architectures, the program body to be executed on the microprocessor remains as C code. Computation performed on satellites (this kind of computation will be referred to as kernel in the rest of the paper) is encapsulated in a procedure call and described in a high-level (currently in C++) intermediate form.

Our goal is to generate efficient and compact configuration and interface code from the C++ intermediate form based on the rest of the program structure and the underlying architecture. It is achieved by following the steps presented on the right of Figure 3. Based on the kernel structures specified by the intermediate form, the baseline code is generated. The architecture and the application program are then used to trigger a set of optimizations to improve upon the baseline code. We will first introduce the concept of the intermediate form in the remainder of this section. The basic code generation process is presented in Section III.2 and optimizations are discussed in Section III.3 and III.4.

The intermediate form is based on the concept of processes (satellites) and queues (connection between satellites). Since the intermediate form has all satellite functionality and connectivity information (shown in Figure 4), it is equipped with configuration and interface code generation capability. A code generation library (corresponding to each satellite as well as the whole kernel) is provided to serve the purpose of automatic code generation. The library currently contains all the basic building blocks (satellites) in Pleiades: MAC/MUL, ALU, SRAM, AGP (address generator), INPUT\_PORT and OUTPUT\_PORT etc. The kernel specified in the intermediate form is like a procedure call – the structure is fixed for each kernel but each invocation of the kernel can pass in different parameters.

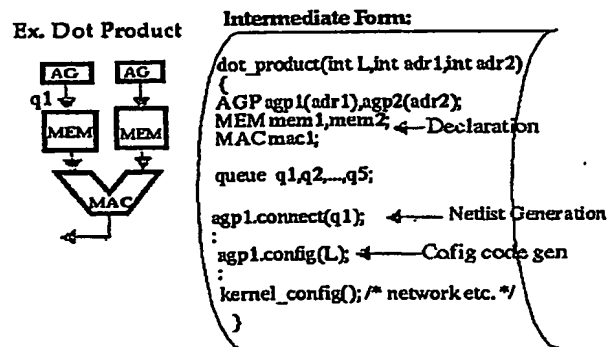


Figure 4 Kernel Mapping and its Specification in the Intermediate Form

## 2. Basic Code Generation Techniques

If the configuration and interface code can be determined at compile time, it is defined as static, while if it can only be determined at runtime, it is dynamic. It is more efficient to execute static code. Based on functionality, the different types of configuration code are divided into three categories. Each category has its unique characteristics that help us determine if it favors static or dynamic configuration:

- **Interface code:** This code takes care of synchronization between the processor and satellites, including resetting of all satellite processors and network before kernel execution. Due to memory and performance constraints, an off-the-shelf operating system for the microprocessor is not used. Instead, a more streamlined version of the operating system is generated. These codes are determined statically.
- **Configuration code for the satellites:** For a given kernel, most of the operations performed on satellites are known, and therefore are statically generated. For example, the configurations of SRAM, ALU and MAC/MUL are fixed for each kernel. Only the configuration of AGP varies from one invocation of kernel to another, and therefore has to be passed in as parameters from the main application program. Hence, configuration in the former case is static and in the latter dynamic.
- **Configuration code for the reconfigurable interconnects:** It is very time consuming to perform runtime routing. Therefore, the netlist is routed on the underlying architecture using the router described in [10] at compile time. Therefore, the configuration is static.

While for a single kernel mapping, the baseline code sequence is determined according to the above basic criteria, generating efficient, yet modular, interface and configuration code for kernels embedded within a larger application involves many more tradeoffs and optimizations which we will describe in the following sections.

## 3. Trade-off and Local Optimizations

A couple of tradeoffs are carefully evaluated to produce a good executable code sequence. First, while minimizing total power consumption is the principal goal, DSP and wireless systems have limited memory size, which constrains the total code size of the application program. We analyze the tradeoff between the two factors – application performance and code size, to determine a right modularity for the generated code. The second trade-off is the reusability of the code versus performance/power. One must customize configuration code for different kernels and applications to achieve satisfactory performance while balancing the development effort and code complexity. Note that in this paper, we translate the optimization of reconfiguration power consumption and performance into the minimization of total number of reconfiguration cycles on the microprocessor.

**Performance/Power vs. Code Size trade-off.** Intuitively speaking, a program that takes the fewest number of cycles to execute would be a “flat” program,

meaning that no modularity is present and therefore kernel procedure calls are in-lined and expanded (in loops). However, a simple application has thousands of kernel procedure calls and each kernel has up to 100 lines of straight code. Clearly, the sheer code size makes this approach infeasible and hence some modularity in the executable code is required. In most digital signal processing and communication applications, there are only limited number of kernel types (dot\_product and FIR in speech coding, DCT in image coding etc.) with different parameters (vector length, starting and ending address in memory). Therefore, we made each kernel type its own procedure call in the generated code. Within a kernel, dynamic and static interface and configuration code of the individual satellite is simply straight code expansion to avoid procedure call overhead.

By following the above procedure, we were able to keep the memory requirements of the program manageable without significantly sacrificing much of the power/performance. For example there are totally about 100 thousand kernels in VSELP. The code size is decreased by four orders of magnitude while the performance is compromised by 50% compared to a flat code sequence (see Table 1). There are certainly other points in the performance versus code size curve but we will mainly concentrate on code optimizations specific to reconfigurable systems in this paper.

Code generation schemes	Normalized Configuration Time (total # of cycles)	Normalized code size
Flat	1	1
Modular	1.56	$7.63 \times 10^{-3}$

Table 1. Comparison of memory and performance between the flat and modular code sequence for the VSELP Application

**Generality vs. Performance/Power tradeoff.** The second tradeoff is code generality vs. performance/power. It is desirable to have customized code routines to take advantage of the special properties of the target application and the kernels involved. However, we also want to maximize code reusability. Significant effort altering the existing library should not be required to achieve acceptable performance.

System bottleneck analysis indicated that address generator configuration is the dominating factor during kernel configuration. This could be explained by the fact that its configuration has the most dynamic component. For a given kernel, only parts of the AGP configuration register fields are relevant and the rest could be treated as don't care. Thus it is advantageous to provide customized AGP configuration routines for the different kernels.

The generality vs. performance/power trade-off issue is also present when we are dealing with the communication from the satellites to the microprocessor. In order to handle applications with multiple parallel threads, we provide the interrupt driven communication primitive in our interface library. That is, the coprocessors

interrupt the embedded microprocessor when they want to communicate. This way communication in a multi-threading environment could be orchestrated in a clean and organized way. Note that sophisticated communication primitives such as interrupt handling and polling do have significant overhead. While for applications that have lots of parallelism to exploit this overhead might be necessary. For applications that are sequential in nature, this expensive overhead cost might not be worthwhile to pay. In such case, simpler and cheaper primitive might be more suitable. For example, the VSELP algorithm is mostly sequential, so the sleep and wakeup power saving mode of the processor is sufficient to implement the communication primitive. In this simply primitive, the microprocessor goes to sleep after it finishes configuration and triggers Pleiades to start running, then Pleiades wakes the microprocessor up when it is done. By replacing the interrupt scheme by the sleep-and-wake communication primitive, significant savings are achieved.

#### 4. Global Optimizations

Besides performing optimizations at the kernel level by choosing the right modularity and providing optimized code libraries, we also exploit the special features of the underlying reconfigurable architecture and the application structure to generate better code sequence.

**Program Cache.** Multiple-context FPGA [12] has been proposed as one of the architecture solutions to reduce configuration cycles in between reconfigurations. Our underlying architecture also supports limited multiple-context configurations. In particular, the AGP instruction registers are deeper and can support up to 5 contexts. Since kernels within DSP applications have only limited instruction patterns, all AGP instructions are often stored in the reconfigurable satellites without reconfiguration from the processor.

**Partial Reconfiguration.** When two identical kernels are called sequentially, only part of the configuration data has to be loaded into the satellites. Specifically, since a particular kernel has a fixed satellite cluster structure and fixed operations, the only configurations that have to be reloaded are the interface code and AGP configurations (for possibly different starting and ending address, vector length etc.). We use the SUIF [13] compiler front-end to discover if a kernel procedure is in a nested loop or identical kernels appear consecutively in the same control flow.

#### 5. Example

To better illustrate the global and local optimization techniques mentioned above, the dot product kernel will be used as a simple example (see Figure 4). The following figure shows the pseudo code for the configuration routine of the kernel.

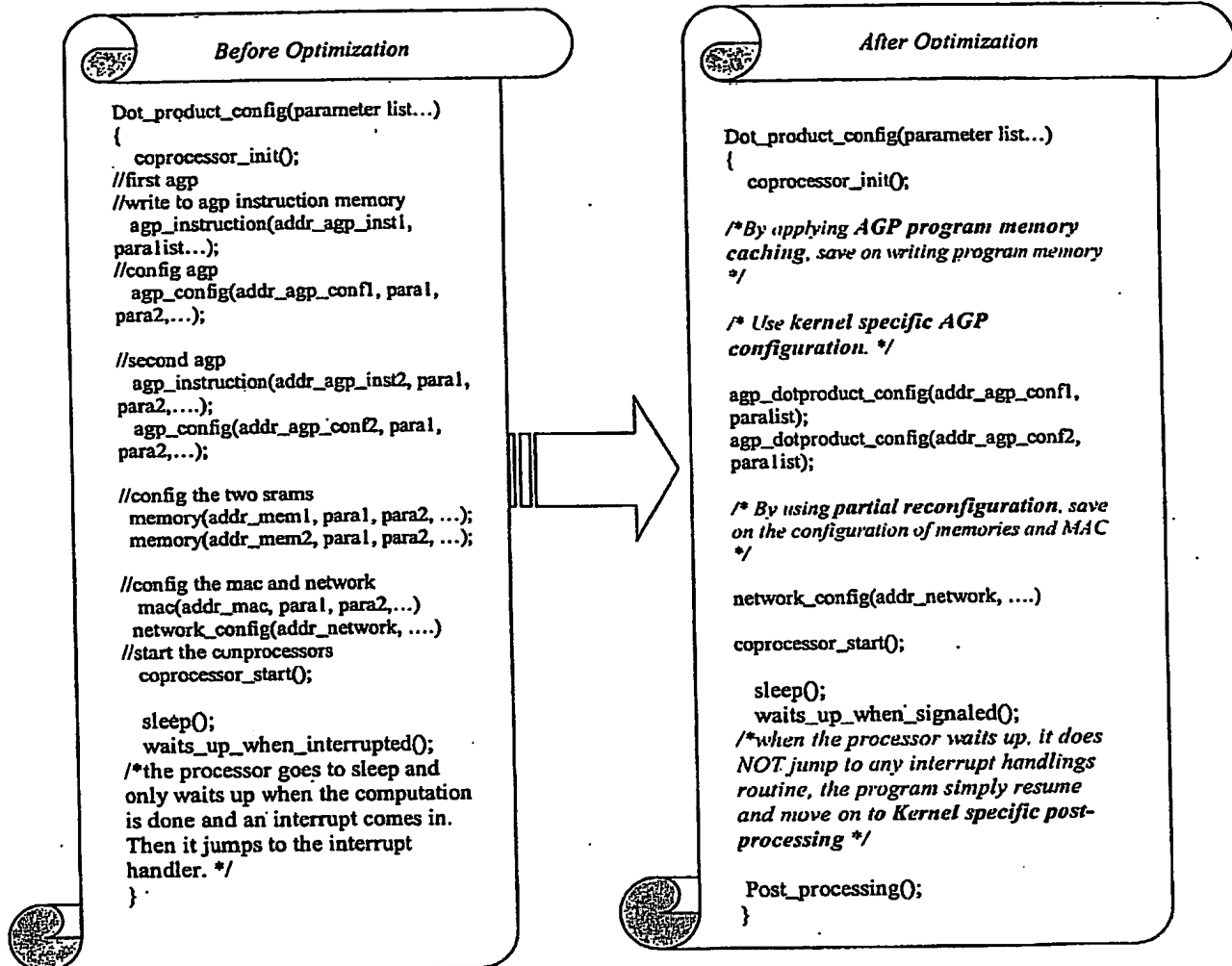


Figure 5. Example kernel program

#### IV. RESULTS

In this section, we will present the evaluation of our code generation process for a speech coding algorithm implemented on a reconfigurable DSP architecture (with an embedded ARM8 [14] as the microprocessor) called Maia. A block diagram of the Maia architecture, which targets baseband source and channel coding algorithms, is shown in the following Figure. The application is the 16-bit

VSELP encoder [15]. The total number of cycles required by VSELP is 126M while it runs entirely on ARM8. After All kernels in the VSELP algorithm have been selected and mapped to the satellite [10], 18.9M cycles remain on the microprocessor (not including configuration cycles).

#### Before optimization

This section shows the result before any optimization is applied. Table 2 gives the configuration cycles required to perform all kernels on the satellites. The total cycle count (48.6M) is significantly smaller than the original one but the fact that more time is spent on configuration than computation is not very satisfactory.

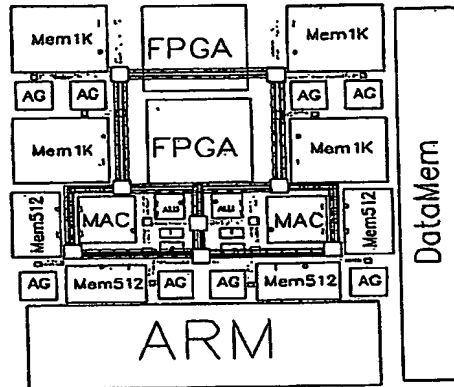


Figure 6 the Maia Architecture

Kernels	Total # of cycle per kernel	# Invocation	Subtotal # of configuration cycles in VSELP
FIR	365	4584	1663992
VSMUL	294	2358	688536
IIR	456	250	113500
Dot_Product1 (1 Mac, 1 AGP)	228	7880	1780880
Dot_Product2 (2 Mac, 2 AGP)	302	83802	25140600
Compute_Code	365	997	361911
<b>TOTAL</b>			<b>29749419</b>

Table 2: TOTAL VSELP cycle breakdown per kernel *before* optimization.

Column 2 of Table 2 shows the cycle count per kernel, column 3 shows the number of times each kernel is called in VSELP, column 4 is the product of column 2 and 3.

### Optimization process

We then proceed to apply the local and global optimizations to improve the configuration time. The following table shows the performance gain (% decrease in configuration cycles) in individual kernels by applying a series of optimization techniques.

<i>Kernels</i>	<i>% saving by applying Kernel specific AGP configuration</i>	<i>% saving by applying Kernel specific post processing</i>	<i>% saving by applying AGP program caching and partial reconfiguration</i>
FIR	32.33%	15.07%	9.86%
VSMUL	29.93%	18.71%	8.16%
IIR	32.46%	12.06%	10.53%
Dot_Product1 (1 Mac, 1 AGP)	19.30%	25.88%	5.26%
Dot_Product2 (2 Mac, 2 AGP)	29.14%	19.54%	7.95%
Compute Code	22.74%	15.07%	9.86%

Table 3: Performance saving in individual kernels by applying a series of optimization technique.

Column 2 shows the % decrease in configuration cycles for each kernel type by utilizing Kernel specific AGP configuration. Column 3 shows the % decrease by applying Kernel specific post processing. Column 4 shows the percent decrease by using AGP program caching.

The following graph illustrates the decrease of number of cycles spent on reconfiguring each kernel type in VSELP with each optimization technique. It also shows the overall decrease of the number of reconfiguration cycles in the VSELP application. We can see that the number goes from 29749419 cycles before optimization to 10818115 cycles after optimizations — a threefold reduction in reconfiguration cycles.

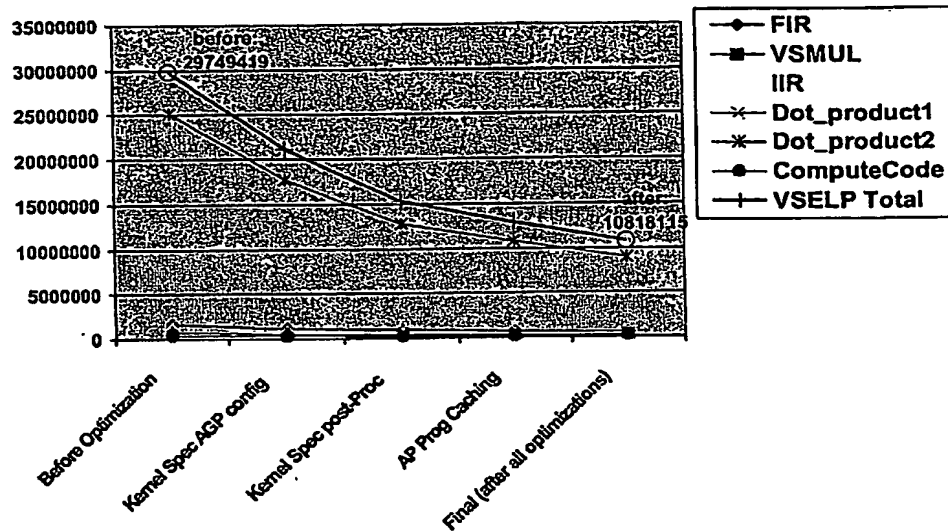


Figure 7. The reduction of reconfiguration cycles with the application of optimization technique

## V. CONCLUSION AND DISCUSSION

In this paper, we presented a code generation process and a set of optimization techniques for reconfigurable architectures. As demonstrated in the paper, a naïve approach could easily generate code that have a high memory demand and is inefficient. In such a case, the overhead of reconfiguring will become a bottleneck of the system. Hence, the code generation process is a very important part of the reconfigurable system design. By carefully considering different system tradeoffs and applying a set of local and global optimization techniques, we are able to improve the performance by a factor of three and greatly reduce the memory requirement.

At present, we addressed the software solution to the code generation process. While one can always write better code, we believe that approaching the problem from the hardware side could push the system performance to a level which software alone can not achieve. Therefore, in our future work, we will also look at the hardware improvement for reconfiguration. Furthermore, we have not considered the overlapping of the ARM processor and the coprocessors. This is due to the sequential nature of our case study application. But in the future, parallelism will be explored to achieve the optimal system performance.

## Acknowledgements

The authors would like to acknowledge DARPA's support of the Pleiades project ( DABT-63-96-C-0026). We also would like to acknowledge the Pleiades Maia design team.

## References

- [1] G. R. Goslin, " A Guide to Using Field Programmable Gate Arrays for Application Specific Digital Signal Processing Performance", Proceedings of SPIE, vol. 2914, p321-331.
- [2] A. Abnous et al, " Evaluation of a Low-Power Reconfigurable DSP Architecture", Proceedings of the Reconfigurable Architecture Workshop, Orlando, Florida, USA, March 1998.
- [3] M. Goel and N. R. Shanbhag, "Low-Power Reconfigurable Signal Processing via Dynamic Algorithm Transformations (DAT)", *Proceedings of Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, November, 1998.
- [4] T. Garverick et al, NAPA1000, <http://www.national.com/appinfo/milaero/napa1000>
- [5] R. Razdan, K. Brace, M. D Smith, "PRISC software acceleration techniques", Proceedings 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors, Cambridge, MA, USA, Oct. 1994
- [6] J. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In J. Arnold and K. L. Pocek, editors, Proceedings of IEEE Workshop on FPGA for Custom Computing Machines, Napa, CA, April 1997.
- [7] A. Abnous and J. Rabae, "Ultra-Low-Power Domain-Specific Multimedia Processors", Proceedings of the IEEE VLSI Signal Processing Workshop, San Francisco, California, USA, October 1996
- [8] S. Hauck, "Configuration Prefetch for single context reconfigurable coprocessors", Proceedings of 1998 International Symposium on Field Programmable Gate Arrays, Monterey, CA, USA, 22-25 Feb. 1998
- [9] Pleiades web page, [http://bwrc.eecs.berkeley.edu/Research/Configurable\\_Architectures/configurable\\_architectures.htm](http://bwrc.eecs.berkeley.edu/Research/Configurable_Architectures/configurable_architectures.htm)
- [10] M. Wan, D. Lidsky, Y. Ichikawa, J. Rabae, "An Energy Conscious Methodology for Early Design Exploration of Heterogeneous DSPs", Proceedings of the Custom Integrated Circuit Conference, Santa Clara, CA, USA, May 1998.
- [11] H. Zhang, M. Wan, V. George, J. Rabae, "Interconnect Architecture Exploration for Low Energy Reconfigurable Single-Chip DSPs" Proceedings of the WVLSI, Orlando, FL, USA, April 1999.
- [12] Sanders, a Lockheed Martin Company, "The Design and Implementation of a Context Switching FPGA", Proceedings of FCCM 1998.
- [13] Stanford Unified Intermediate Form, <http://www-suif.stanford.edu/suif/>
- [14] Advanced RISC Machines, ARMulator version 2.10
- [15] I. Gerson and M. Jasiuk, "Vector Sum Excited Linear Prediction (VSELP) Speech Coding at 8Kbps, Proceedings of the International Conference on Acoustic, Speech, and Signal Processing, pp. 461-464, April 1990.

# Reconfigurable Processing: The Solution to Low-Power Programmable DSP

Jan M. Rabaey

Department of EECS, University of California at Berkeley

## Abstract

One of the most compelling issues in the design of wireless communication components is to keep power dissipation between bounds. While low-power solutions are readily achieved in an application-specific approach, doing so in a programmable environment is a substantially harder problem. This paper presents an approach to low-power programmable DSP that is based on the dynamic reconfiguration of hardware modules. This technique has shown to yield at least an order of magnitude of power reduction compared to traditional instruction-based engines for problems in the area of wireless communication.

## 1. Introduction

Keeping the power dissipation within bounds is rapidly becoming one of the main challenges in contemporary digital design. This is especially the case in the domain of wireless communications. For compelling business reasons, extending the time between battery recharges has long been one of the highest priorities in the design of the mobile terminals. Energy-efficient implementations combining dedicated hardware with specialized processing elements have been proposed so that in-use and standby times of terminals are now measured in the range of hours and hundreds of hours, respectively.

There has been much ado recently about the need for "multimodal" or "software" radios, where most of the functionality of the radio is relegated to software. This concept has the advantage that a single terminal can support multiple standards (e.g. DECT and GSM) or can adapt dynamically to the requested services (voice, low-rate or high-rate data). The implied programmable approach annihilates many of the power-reduction techniques used in traditional terminals.

Minimizing the power dissipation has also become an issue for the basestation, where a large array of channels are processed simultaneously. It is desirable for a single basestation module to support multiple standards (e.g. GSM, IS95, DOCOMO), which by necessity leads to a programmable implementation platform. The higher energy dissipation associated with the programmable solution combined with the high computational requirements and the inaccessible and distributed locations of the basestations provide for a strong argument to keep power dissipation minimal there as well.

All these arguments point to the need of powerful programmable low-power compute engines. Numerous approaches towards low-power processing for wireless applications have been proposed: general-purpose (GP) processors with extended instruction sets; GP + co-processor structures; application-specific processors, and proces-

sors with power-scalable performance. This paper proposes an alternative approach to power-minimization based on a dynamic matching between architecture and computation. It will be demonstrated that reconfigurable architectures are an excellent vehicle for accomplishing this goal assuming that the granularity of reconfiguration is chosen in accordance with the model of computation of the targeted task.

## 2. Granularity of Programming Model

The term "programmable" typically has a strong association with the "stored-program" concept as originated from the computer world. A program is typically a set of instructions that dynamically modify the behavior of an otherwise statically connected modules such as memories, registers, and (a) datapath(s). In the last few decades, the concept of programming has gradually been extended to include the dynamic reconfiguration of interconnect networks and logic functionality as well. A first step in this direction was to connect multiple traditional processors in a dynamically adjustable configuration (multi-processor architectures). More recently, "adaptive computing systems (ACS)" have received increasing attention [1]. Typically, adaptive computing systems are constructed from arrays of field-programmable devices (FPGAs) and claim orders-of-magnitude in performance improvement over traditional computers by providing programmability (reconfigurability) at the gate level. While such performance improvements are possible for specific computational kernels, they may come at the price of an increase in area, power dissipation, and programming ease.

Programmability can actually come at multiple levels of granularity, and each has its own preferred and optimal application domain. Some of these levels are illustrated in Fig. 1. The two extremes are the already mentioned stored-program (a) and the gate (or transistor)-level (d) reconfigurable modules. Other options for reconfiguration at the functional-module (b) and the datapath-operator (c) levels.

The difference between the various computational models lies in the granularity of the composing modules, the distribution of the program storage, and the interconnect structure. Especially the latter requires some extra explanation: Stored-instruction engines rely on shared buses for the transfer of data (and instructions); the reconfigurable dataflow architecture uses either complete or reduced crossbar networks of busses; the interconnect network of a reconfigurable datapath exploits the bit-sliced structure and the predominantly one-dimensional flow of data by using an asymmetrical network: reconfigurable busses in one direction and a bit-level

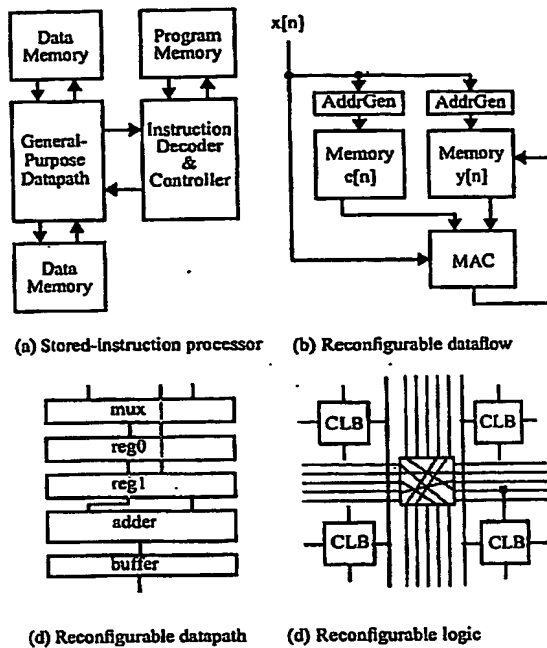


Fig. 1. Granularity of reconfiguration

mesh in the other; reconfigurable logic typically relies on bit-level mesh networks.

While it is theoretically conceivable to map virtually any computational function onto any of the reconfiguration structures, doing so inevitably leads to either area, power or performance penalties. In this paper, we demonstrate that combining the various models into a single architecture can lead to dramatic power reductions in wireless applications (but also for other domains that rely heavily on arithmetic and bit-level computations such as multimedia).

### 3. Granularity and Energy Consumption

The intense research in low-power electronics of the last five years has clearly identified the two key approaches to achieve substantial energy reduction:

- Operate at the lowest possible voltage and frequency. The loss in performance due to the lowering of the supply voltage is off-set by exploiting concurrency in the form of pipelining and parallelism. Observe that most wireless and multimedia applications have ample opportunity to do so as the algorithms in question exhibit an extensive amount of implicit concurrency. It was also observed [2] that the overexploitation of concurrency to reduce the supply voltage can lead to energy penalties: at a certain level of parallelism the overhead circuitry starts to dominate any gains from voltage reduction and the energy/computation starts to increase. This results in an optimum supply voltage and a optimum level of concurrency (Fig. 2).

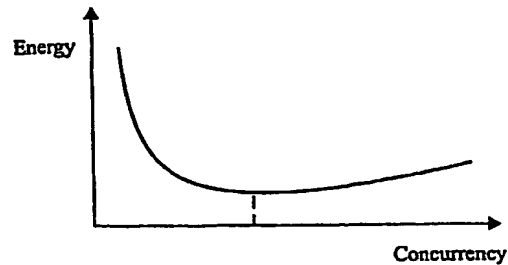


Fig. 2. Energy versus Concurrency

Observe that the position of that optimum is a strong function of the computation at hand. For example, it is perfectly conceivable to implement a complex computational algorithm on an PGA, thus exploiting the bit-level concurrency to the maximum. This approach will most often lead to an energy-inefficient solution as the PGA architecture implicitly carries an extensive energy overhead. On the other hand, for algorithms that require extensive bit-level operations (such as for instance encryption) the PGA architecture provides the most energy-efficient solution.

- Reduce the energy waste. While the former approach has received the most attention due to its quadratic nature, it is ultimately by keeping the energy overhead to a minimum that the most impressive power savings will be made (the reduction of supply voltage is lower-bounded by the implementation technology and the allowable standby power dissipation). Application-specific solutions present the most effective way of reducing energy waste and have been shown to lead to huge power savings [3]. As stated in the introduction, however, programmability is often a desirable and necessary feature. Making an architecture programmable (or reconfigurable) inherently carries with it a large energy overhead which most often dominates the energy dissipated for the intended computation. For instance, less than 15% of the energy dissipated in microprocessors goes to computation (it is actually far less if one accounts for the fact that this number also includes global communication) [4]. A energy/gate switch in an FPGA is approximately 7 to 10 times larger than in an ASIC implementation of the same gate.

A number of techniques can be identified to reduce energy overhead in programmable architectures:

- match computational and architectural granularity. Performing a multiplication on a PGA is bound to carry a huge amount of waste, so does executing a large dot-vector product on a microprocessor.
- preserve the locality inherent in the algorithm. Distributing the storage, interconnect, programming, and computation resources leads to a reduced cost per operation. For instance, executing a single instruction on a processor requires the fetching of the instruction from a large memory, decoding the instruction, and routing the control signals to the datapath.
- energy-on-demand. no unit should ever consume energy if not in use.

- exploit signal statistics, avoid extensive multiplexing of communication and computation resources if data signals exhibit strong temporal correlations.

The choice of the correct programming model can help to enable these power-reduction techniques. In the next paragraph, we introduce a domain-specific architectural template that combines the four models of computation.

#### 4. Multi-granularity Architecture

Fig. 3. presents a reusable template that can be used to implement a domain-specific processor instance, that can then be programmed to implement a variety of algorithms within a given domain of interest [5]. All instances of the architecture template share a common set of control and communication primitives. The type and the number of processing elements may vary; they depend upon the properties and the computational requirements of the particular domain of interest.

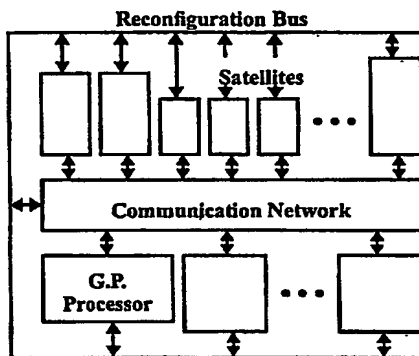


Fig. 3. Multi-granularity architecture template.

The architecture is centered around a reconfigurable communication network. Communication and computation activities are coordinated via a distributed data-driven control mechanism. Connected to the network are an array of heterogeneous, autonomous processing elements, called satellite processors. These could fall into any of the reconfigurable classes: a general microprocessor core (most of the time only one of these is sufficient, a dedicated functional module such as a multiply-accumulator or a DCT unit, an embedded memory, a reconfigurable datapath, or an embedded PGA. Observe that each of the satellite processors has its own autonomous controller, although the instruction set of most of these modules is very shallow. For instance, the programmability of a multiply-accumulate processor is typically restricted to the number of samples to be accumulated, and numeric parameters such as the overflow, rounding and scaling characteristics.

The microprocessor core plays a special role. Besides performing a number of miscellaneous non-compute intensive or control-dominated tasks, it configures the satellite processors and the communication network (over the reconfiguration bus). It also

manages the overall control flow of the application, either in a static compiled order, or through a dynamic real-time kernel.

The application is partitioned over the various computational resources, based on granularity and recurrence of the computational sub-problem. For instance, a convolution is mapped onto a combination of address generator, memory, and multiply-accumulate processors. The connection between these modules is set up by the control processor and remains static during the course of the computation. The same modules can in another phase of application be used in a different configuration to compute, for instance, an FFT.

The architectural template meets all the requirements for low-energy computation, as stated higher: it supports concurrency at multiple levels of granularity; it preserves locality; it employs application-specific modules if needed; it minimizes control overhead; its data-driven synchronization mechanism ensures that modules will only consume energy when needed; and the static configuration approach of the interconnect network preserves signal correlations.

For more information about the architecture and its components, please refer to [5].

#### 5. Examples

The effectiveness of the multi-granularity architecture in reducing energy dissipation will be demonstrated using two examples.

##### 5.1 A voice-coder processor

Low-power voice coders are an essential component of virtually all cellular terminal and basestation modules. Analysis of the computational requirements of a full-rate VSELP voice coder shows that 92% of the computational energy can be attributed to the inner loops of four functions, while vector-dot products account for 65% of the total power. Obviously, doing well on these operations brings us a long way towards a low-power solution. demonstrates how the dot-vector products are mapped to the reconfigurable architecture. The resulting structure consumes only 175 pJoule per MAC operation (including the memory accesses) (at 1.5V in a 0.6 mm CMOS technology). At a maximum clock rate of 30 MHz, this translates in a truly astounding 5.7 GOPS/Watt (with each OP being a full dot-vector operation). The total predicted power-dissipation for a full VSELP processor ranges between 2 and 5 mW.

##### 5.2 CDMA baseband processing

Analysis of the baseband processing in a direct-conversion CDMA receiver (Fig. 5.) reveals that most of the computation resides in the execution of the 9 high-speed correlators [6], each of which multiplies a sequence of 4-bit numbers with a serial bit-stream. The high-performance requirements for the correlator units (Fig. 6.) make it impossible to construct the processor from functional modules, as was done for the dot-vector processor in the previous chapter. While it is conceivable to build a dedicated correlator processor, this unit would only be useful for the implementation of CDMA-based receivers. More appropriate here is the use of a reconfigurable datapath processor, that can also be

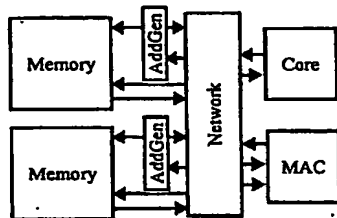
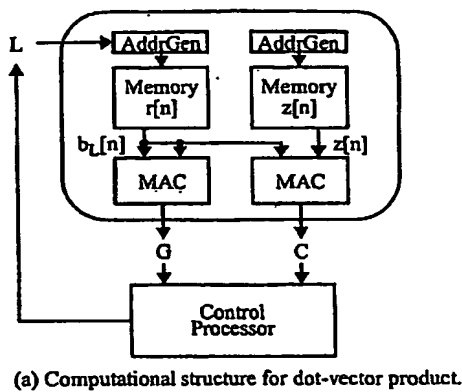
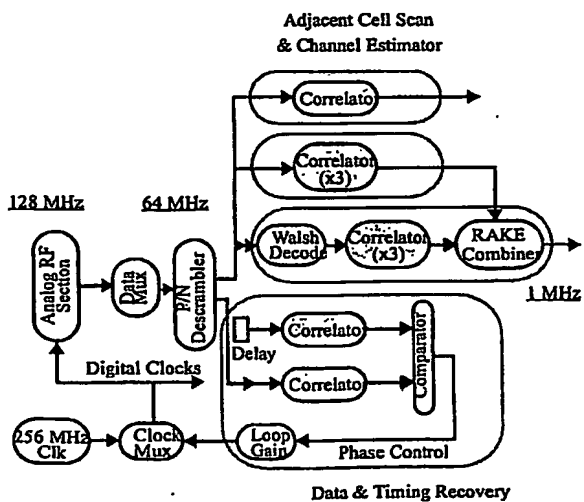


Fig. 4. Energy-efficient implementation of dot-vector product on multi-granularity architecture.



used to implement other computational structures such as filters or modulators.

## 6. Summary

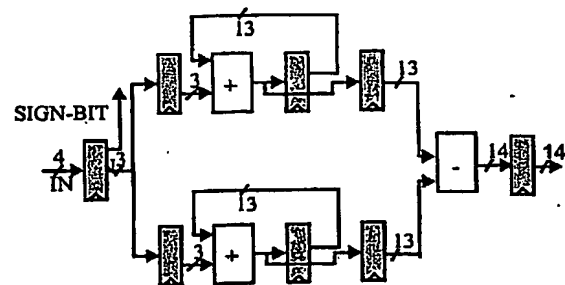
Contrary to the common belief, reconfigurable computing presents a very attractive opportunity high-performance programmable computing with high energy efficiency. Central to the approach is the matching between the granularity of computation in application and architecture. This by necessity leads to heterogeneous domain-specific architectures. This approach lends itself perfectly to the design of multi-modal, soft-programmable wireless terminals and basestations.

## 7. Acknowledgments

This research is sponsored by DARPA as a part of its Adaptive Computation Systems initiative. Also appreciated are the sponsorship of Rockwell, Cadence, Motorola, Sharp and Sony. The author acknowledges the contributions of Arthur Abnous, Marlene Wan, George Varghese, Katsunori Seno, and Yuji Ichikawa to this research.

## References

- [1] *Adaptive Computing Systems*, [http://www.ito.darpa.mil/ResearchAreas/Adaptive\\_Computing\\_Systems/ACSIntro.html](http://www.ito.darpa.mil/ResearchAreas/Adaptive_Computing_Systems/ACSIntro.html).
- [2] A. Chandrakasan, *Low Power Digital CMOS Design*, Chapter 4, Kluwer Academic Publishers, 1995.
- [3] J. Rabaey, *Low Power Digital Design*, in *Circuits and Systems Tutorials*, pp. 373-386, LTP Electronics LTD, 1994.
- [4] R. Gonzalez and M. Horowitz, "Energy Dissipation in General Purpose Processors," *Digest of Technical Papers, 1995 IEEE Symposium on Low Power Electronics*, pp 12-13, San Jose, 1995.
- [5] A. Abnous and J. Rabaey, "Ultra-Low Power Domain-Specific Multimedia Processors", in *VLSI Signal Processing IX*, Ed. W. Burleson et al., IEEE Press, pp. 459-468, November 1996.
- [6] S. Sheng, L. Lynn, J. Peroulas, K. Stone, R.W. Brodersen, "A Low-Power CMOS Chipset for Spread-Spectrum Communications," *1996 International Solid-State Circuits Conference*, Feb 8-10, 1996, San Francisco, CA. Oct. 1995.



# Instruction Generation and Regularity Extraction For Reconfigurable Processors

Philip Brisk

Adam Kaplan

Ryan Kastner

Majid Sarrafzadeh

Computer Science Department  
University of California, Los Angeles  
Los Angeles, CA 90095

{philip, kaplan, kastner, majid}@cs.ucla.edu

## ABSTRACT

The increasing demand for complex and specialized embedded hardware must be met by processors which are optimized for performance, yet are also extremely flexible. In our work, we explore the tradeoff between flexibility and performance in the domain of reconfigurable processor design. Specifically, we seek to identify regularly occurring, computation-heavy patterns in an application or set of applications. These patterns become candidates for hard-logic implementation, potentially embedded in the flexible reconfigurable fabric as special optimized instructions. In this work we present an extension to previous work in instruction generation: an algorithm that identifies parallel templates. We discuss the advantages of parallel templates, and prove the correctness of our algorithm. We introduce an All-Pairs Common Slack Graph (APCSG) as an effective tool for parallel template generation. Finally, we demonstrate the effectiveness of our algorithm on several applications' dataflow graphs, reducing latency on average by 51.98%, without unreasonably increasing chip area.

## Category

1. Compilers and Operating Systems

## General Terms

Algorithms, Performance, Theory

## Keywords

Hardware Compiler, Template, Slack, Control Data-flow Graph

## 1. INTRODUCTION

The complexity of modern computing systems, combined with the creation of faster, smaller circuitry, has fueled the genesis of extremely small and powerful embedded systems. With the increasing design trend toward smaller and more intimate

computing systems comes a need for specialization, as each of these smaller systems is more limited in functionality than a general-purpose processor. Embedded processors need not perform the same set of operations as their larger counterparts, and yet the limited number of computations they perform must be optimized for power, area, and also computational speed. Frequently, ASICs are employed in the creation of such devices, due to their traditional ability to meet these demands.

Yet, ASICs are extremely inflexible; once the device is fabricated, the functionality can never be changed. Thus, as design standards and protocols evolve, embedded ASICs must be thrown away, and new ones built to meet the new demand. This is both wasteful of good circuitry and also extremely challenging to the designers of these chips. These devices should be able to satisfy the rigid performance requirements that ASICs have classically met. Reconfigurable devices (built upon programmable logic device technology) contain logic that can be quickly and completely re-programmed. Thus, due to their inherent flexibility, reconfigurable chips have been proposed as the design platforms of specialized embedded processors. Unfortunately, there exists a tradeoff between the flexibility and performance of a system.

We desire the ability to customize a system towards the special set of tasks that it needs to perform, imbued with the ability to evolve. We believe that, given the subset of functions that such a system would need to execute frequently, it is possible to implement some of these instructions as hard (or fixed) logic blocks. Such blocks would perform better than the softer, reconfigurable parts of the device. If well chosen, they could enhance the performance of the system, providing the benefits of frequent hard-logic execution, while simultaneously affording flexibility to other (possibly subsuming) execution components.

With the goal of extracting instruction regularity, and consequently generating more complex instruction candidates for hard logic implementation, we propose a scheme whereby an application or set of representative applications would be sent to a compiler. Using the compiler's intermediate representation (IR) of a program, we can determine a candidate set of templates (instructions) to optimize. In Section 2, we will further expound upon template generation as an idea, and discuss the control dataflow graph as a compiler IR. In Section 3, we will explain a previous approach to sequential template generation, including some of its shortcomings. In Section 4, we present our

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.  
CASES 2002, October 8-11, 2002, Grenoble, France.  
Copyright 2002 ACM 1-58113-575-0/02/0010...\$5.00.

algorithmic extension to template generation. In Section 5, we discuss our implementation as well as some preliminary empirical results of our instruction generation algorithm. In Section 6, we present some related work. Finally, in Section 7, we conclude with some future direction for this work.

## 2. TEMPLATE GENERATION

Templates are repeated occurrences of possibly interdependent nodes and edges in a dataflow graph (DFG). Each node in a DFG represents some simple operation, such as ADD or LOAD. Templates can be thought of as vertex clusters, or super-nodes, which represent instructions at the architecture level. In this paper, we define two different types of templates. Sequential templates correspond to nodes connected by directed edges in the DFG (i.e. nodes which represent a sequential execution operation). Parallel templates correspond to nodes whose operations can be scheduled for simultaneous execution. Sequential templates contain direct data dependencies (as they follow DFG edges) and thus cannot directly increase inherent parallelism in the execution. Parallel templates have no data dependencies between them, but may restrict the scheduling mechanism at lower synthesis stages.

Template generation attempts to extract regularities of simple mathematical operations on data flow graphs (DFGs). DFGs define interdependent sequences of ALU operations that occur between branching statements. DFGs are an integral piece of an intermediate compiler format known as control data flow graphs (CDFGs). CDFGs are a commonly accepted form of internal representation upon which a compiler can perform optimization and template matching. The nodes in CDFGs are basic blocks, sequences of arithmetic instructions with exactly one entrance (branch target) and exactly one exit (branching or halt instruction). An example of a CDFG is presented in Figure 1.

The template generation algorithms we discuss operate on DFGs, but do not yet extend across control nodes. Template generation is performed on the DFGs within each CDFG node, but branch-subsuming templates have not been investigated yet.

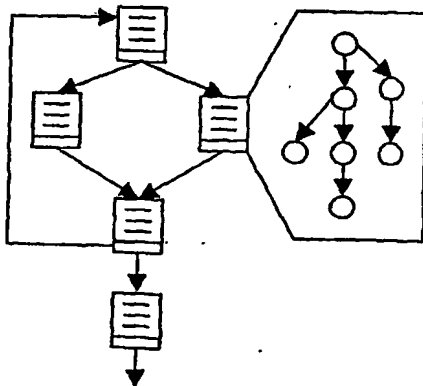


Figure 1. A Typical CDFG

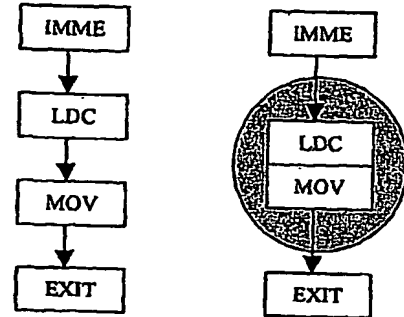


Figure 2. An example of a DAG with and without a sequential template.

## 3. SEQUENTIAL AND PARALLEL TEMPLATES

Sequential instruction generation was first applied to reconfigurable systems by Kastner [1]. Sequential templates attempt to extract sequential regularity in a DFG. Sequential regularity can be observed in a DFG when multiple instances of directed edges connect nodes of type A to nodes of type B (A and B correspond to simple ALU operations). Determining a set of candidate sequential templates is simple. One simply needs to examine all of the edges in a DFG and count the different types that occur. An example of a DFG with and without a sequential template is shown in Figure 2.

Since we wish to extract regularity, the edges that occur the most often will be the best candidates for sequential templates. The count for any given edge type indicates that a particular data dependency between two operations occurs frequently within the data flow of a program; however, the count alone is a misleading heuristic. Often, candidate templates will overlap one another, hence only some subset of candidate edges can legally become templates at any given time.

In Figure 3, we observe two sequential edges of type (CVT,XOR). Unfortunately, both of these edges are incident on the same XOR-node. Therefore, we cannot cluster both edges because they overlap at the XOR-node; however, we can cluster one or the other of the edges. In the future, we will consider the effects of clustering all three of these nodes into a larger template.

When it is time to actually create the template, a super-node replaces each node-edge-node combination. The super-node will have the same connectivity as the node-edge-node, and will maintain the node-edge-node information internally.

Before we can cluster nodes, we must traverse the DAG and record the number of occurrences of particular edge types. Consequently, we will cluster the most frequently occurring edge type(s).

Sequential clustering iterates between the traversal and clustering phases until some stop condition is met. Without a stopping condition, all nodes and super-nodes would eventually be clustered to the point where the DAG consists of only one node, which is not a desirable outcome. There are several potential stopping conditions. Kastner [1] experimented with different

stopping criteria, using the amount of graph coverage as a metric of success.

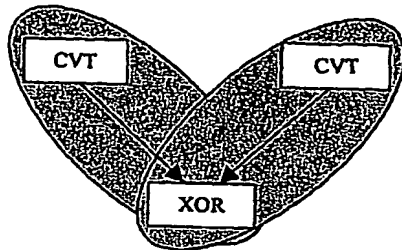


Figure 3. An example of a DAG with conflicting candidate sequential templates

#### 4. PARALLEL TEMPLATES

Parallel templates attempt to extract forms of regularity that are free of data dependencies. Parallel regularity cannot be observed directly by examining edges in a DFG. Alternatively, one must examine DFG nodes in an order that is orthogonal to the flow of data in the DFG. An example of a DFG with a parallel template is shown in Figure 4.

Two nodes are candidates to be parallel templates if they could possibly be scheduled at the same time step by some scheduling heuristic; or equivalently, if the common slack between the nodes is greater than zero. The slack of a DFG node can be loosely described as the number time steps at which the node could be scheduled without violating data dependencies or increasing the length of the critical path of the DFG. The common slack shared by any pair of nodes in a DFG is the number of time steps at which their operations could execute simultaneously within data dependency constraints.

In order to formally determine which nodes may be scheduled together, we create an All-Pairs Common Slack Graph (APCSG) from the DFG. Every pair of nodes in the graph is considered. If the common slack between the two edges is greater than zero, an edge weighted with the common slack is added between the two nodes in question.

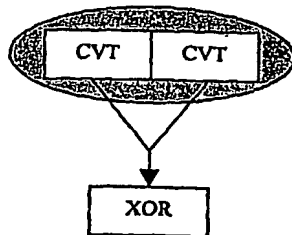


Figure 4. The DAG in Figure 3 with a parallel template.

Calculation of common slack between two nodes assumes that the graph has been topologically sorted in advance using ASAP scheduling. The level of a node is defined as the earlier possible

time step at which it is scheduled during ASAP scheduling. The slack of a node may be calculated as the difference between its level sorted by ALAP scheduling and its level sorted by ASAP scheduling.

The edges in the APCSG represent the set of every pair of DFG nodes which can be scheduled at the same time step by some scheduling heuristic. Therefore, we have chosen the nodes adjacent to the set of APCSG edges to represent our candidate set of parallel templates.

A more generalized model would take note of the fact that any clique of size  $k$  in the APCSG is a set of  $k$  nodes that can be scheduled at the same time step by some scheduling heuristic. The set of APCSG edges is simply the set of all two-node cliques in the APCSG. Nonetheless, we chose only the nodes adjacent to edges in the APCSG as our candidate set of parallel templates because the clique problem is NP-complete. Examining the costs and benefits of the clique finding approach is promising future work.

To determine which types of parallel nodes should be clustered, we count the number of APCSG edges between each node type, taking into account the common slack values. Then, we cluster the corresponding DFG nodes whose APCSG edges have the highest count.

An example of a DFG and its corresponding APCSG are shown in Figures 5 and 6.

In an analogous manner to the algorithm for sequential template generation, the parallel clustering algorithm selects the most frequently occurring edge type in the APCSG. Unlike the sequential methodology, however, common slack is also used to determine exactly which type of edges should be used for contraction. For each type of edge that appears in the APCSG, a weighted sum is taken.

$$\text{Weight}(\text{parallel edge type } e) = \sum_{i \in \text{instances}(e) \text{ in APCSG}} \text{APCSG\_Weight}(i)$$

Once the weighted sums have been computed, we cluster edges of the type that has the greatest weighted sum. Among all edges of the chosen type, we will first cluster those with the largest common slack values. Once again, we justify this heuristic by arguing that clustering nodes with large common slack values will minimize the flexibility that we lose after clustering the nodes. This leads to greater amounts of regularity that can be extracted in future iterations of the algorithm. This makes it an intuitively favorable heuristic.

Initially, we prefer to cluster those edges between nodes with large common slack values. This helps to reduce the amount of slack in the DFG that is lost as a result of clustering the nodes. By similar logic, this heuristic reduces the number of edges that must be removed from the APCSG. Hence, it preserves parallelism in the DFG.

In the case of our example in Figures 5 and 6, the most frequently occurring weighted edge type is (MUL, MUL), which occurs three times with weighted sum of 4. The edge with weight 2 is chosen for clustering. The resulting DFG and APCSG are shown in Figures 7 and 8.

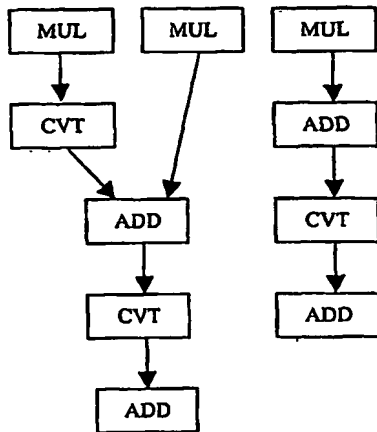


Figure 5. An example DFG that will be used to illustrate the creation of the APCSG.

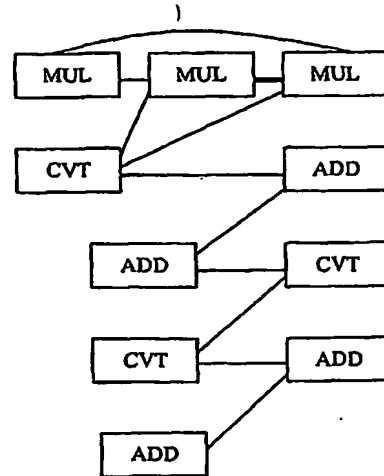


Figure 6. The APCSG. All edges have weight 1, except for the bold edge, which has weight 2.

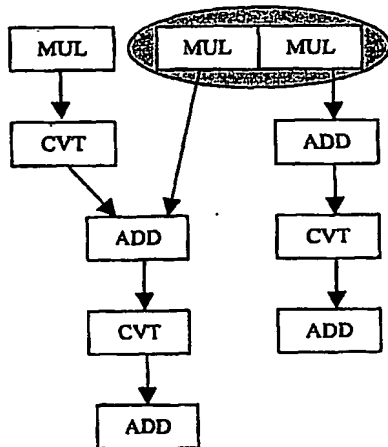


Figure 7. The updated DFG after one round of parallel clustering.

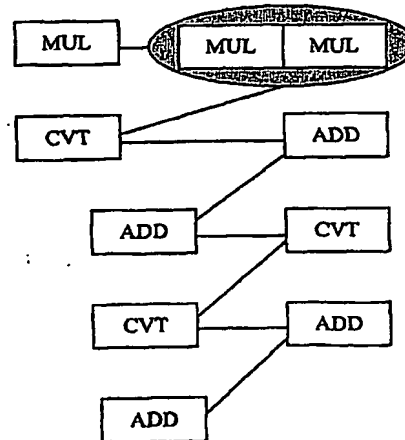


Figure 8. The APCSG after one round of parallel clustering.

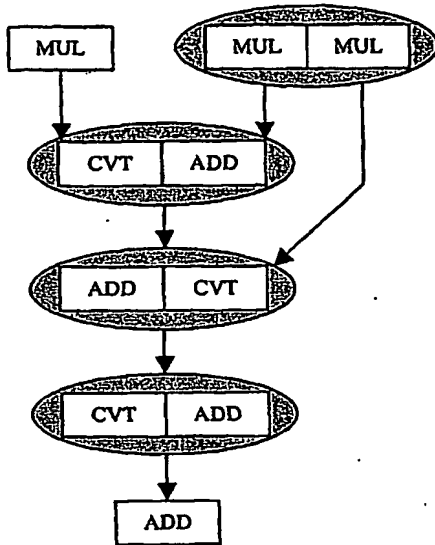


Figure 9. The updated DFG after two rounds of parallel clustering.

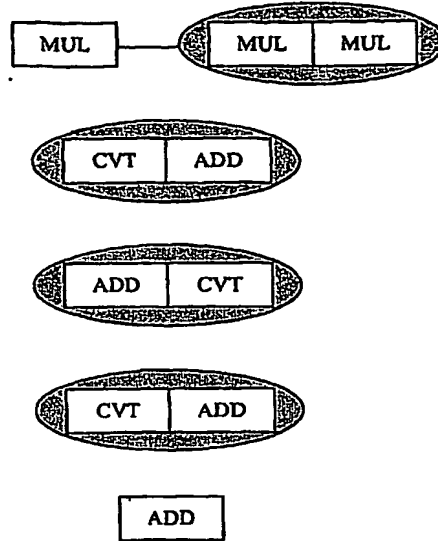


Figure 10. The APCSG after two rounds of parallel clustering.

The most frequently occurring edge type in Figures 7 and 8 is (CVT, ADD), so the corresponding nodes are chosen for clustering. Figures 9 and 10 show the resulting DFG and APCSG after one final iteration of clustering.

#### 4.1 Combining Sequential and Parallel Template Generation

The processes of determining candidate sets of sequential and parallel templates are inherently different. Sets of sequential candidates can be deduced by examining edges in the original DFG; whereas sets of parallel candidates can be found by examining the edges in the APCSG. Candidate sets for sequential and parallel templates can be determined independently of one another, but they can be treated in the same way. If we use the weighted sum heuristic, we can combine parallel and sequential template generation into a single algorithm. At present, we equate number-of-occurrences of a given edge type with its summed slack values. (In the future, an empirically-driven modification to this cost function may be considered.)

The algorithm for simultaneous sequential and parallel template generation is shown in Figure 11. The profiling functions determine the most frequently occurring edge types in the DFG and APCSG respectively. The `dispatch_clustering` function determines whether the nodes being clustered are sequential or parallel, and then dispatches the information to the appropriate clustering function.

The complexity of construction of the APCSG is  $O(V^2)$  because each pair of vertices in the DFG must be examined. The complexity of the profiling functions are  $O(E_{dfg})$  and  $O(E_{apcsg})$  respectively because each edge need only be examined once. The number of iterations of the while loop depends purely on the stopping heuristics; however, in the worst case, only one pair of nodes will be clustered during each iteration until there are no edges left. Therefore, the overall complexity of the template generation algorithm is  $O(E_{dfg}^2 + E_{apcsg}^2)$ .

1. Given a labeled DFG  $G(V, E)$
2.  $G' \leftarrow \text{construct\_APCSG}(G)$
3.  $\#C$  is a set of edge types
4.  $C \leftarrow \{\}$
5. While not stop\_conditions\_met( $G$ )
  - a.  $C \leftarrow \text{profile\_graph}(G) \cup \text{profile\_graph}(G')$
  - b. `dispatch_clustering`( $G, G', C$ )

Figure 11. Algorithm for Simultaneous Sequential and Parallel Template Generation

## 4.2 DAG Isomorphism

An important issue arises when comparing templates to one another to determine regularity. We wish to find, at every stage, the number of occurrences of a given edge type. It is mandatory to check both nodes of each edge for equivalence. If both nodes are equal to those of the type we wish to locate, then we have found another occurrence of this edge type.

As clustering commences, the vertices attached to each edge may become harder to compare. These vertices may be super-nodes, combinations of two or more original nodes with an internal directed acyclic graph (DAG) representation of the original node set. Our approach to DAG isomorphism must also consider the type of operation represented by each node. All isomorphic patterns must match node types as well as nodes and edges.

In general the test for isomorphism cannot be applied efficiently for directed acyclic graphs. Although DAG isomorphism has never been proven NP-complete, all proposed solutions to the problem possess exponential running time. We thwarted this problem in our work by integrating a polynomial-time approximation of DAG isomorphism into our implementation, via the University of Naples' VFLib Graph Matching Library [2]. Although this approximation is not perfect (i.e. it will miss some DAGs that are isomorphic to one another) it is acceptable for our purposes.

## 5. EXPERIMENT AND RESULTS

We implemented our algorithms on top of the Stanford SUIF compiler [3]. Building upon some of the modifications made by Kastner for sequential template construction [1], we added a full implementation of the APCSG and its generation, as well as the code to successfully merge parallel and sequential templates. Finally, we added the higher-level heuristic that performed the combination of sequential and parallel clustering. Our template sizes were restricted to five internal nodes, and our algorithm terminated when the total number of super-nodes (clustered vertices) in the DFG was less than half of the original number of vertices.

Our initial goal was to determine how template generation would affect the general scheduling of instructions, regardless of whether the target of compilation is a super-scalar pipelined architecture or the synthesis of new hardware. Although application synthesis is not the goal of the compiler, a high-level synthesis tool could perform scheduling of the resulting clustered DFGs.

Specifically, our experiment is designed to determine the scheduling latency of our generated DFGs (intuitively analogous to the time of execution on a powerful processor). We compare this latency to the scheduling latency of the original (non-clustered) DFG. Assuming that latency is generally improved by the addition of special blocks of logic to execute regular instructions, we furthermore wish to explore the impact that these clustering decisions will make on chip area. Specifically, a thorough exploration of the latency/area tradeoffs of clustering is required in order to evaluate our methods.

In order to simulate the results of our algorithm, we compiled and generated instructions for four programs: an image convolution algorithm [4], DeCSS (the decryption of DVD encoding) [5], the DES encryption algorithm [6], and the Rijndael AES encryption

algorithm [7]. These algorithms are typical candidates for industrial hardware implementation (as cameras, DVD players, and embedded encryption devices must perform these operations). Additionally they are computationally intensive, leading to generally large DFGs which are benign to regularity extraction. From each compiled CDFG of the programs, four representative DFGs were selected for scheduling. The scheduling algorithm we used has been described in detail in [8], and is comparable to the state-of-the-art in the research community. The resulting latency of each scheduled DFG (both with and without clustering) is recorded in Table 1. In Table 2, we record both the decrease in latency and the increase in FPGA area that resulted from our clustering algorithm.

Clearly, clustering of DFGs reduces the number of total instructions, and increases the potential to execute frequently occurring sets of parallel operations. This directly improves the schedule of the application DFGs, demonstrating latency improvement by as much as 76.19% on our largest DFG (the first basic block of the DES encryption algorithm: 150 nodes). For every DFG scheduled, latency was improved by at least 25%, a surprisingly good figure. Additionally, the FPGA area increased an average of 21.55% (maximally 150% in some smaller DFGs). Occasionally, even decreased area was realized via clustering, presumably due to improved utilization of regular specialized components. Overall, the average latency improvement (51.98%) shadowed the area gains (average 21.55%), especially on larger, more complex DFGs.

		No Clustering	Sequential and Parallel Clustering
Convolve	Node 1	10	5
	Node 2	6	4
	Node 3	8	2
	Node 4	8	6
DeCSS	Node 1	11	6
	Node 2	10	3
	Node 3	56	31
	Node 4	21	9
DeS	Node 1	84	20
	Node 2	59	24
	Node 3	20	11
	Node 4	18	11
Rijndael AES	Node 1	24	15
	Node 2	56	32
	Node 3	17	6
	Node 4	6	2

Table 1. Latency Measurements for Each Scheduled DFG (in CPU Cycles)

		Size of original DFG (nodes)	% Latency Decrease	% FPGA Area Increase
Convolve	Node 1	20	50.00	66.67
	Node 2	13	33.33	-4.55
	Node 3	17	75.00	31.25
	Node 4	19	25.00	4.17
DeCSS	Node 1	21	45.45	150.00
	Node 2	13	70.00	-12.50
	Node 3	121	44.64	25.00
	Node 4	55	57.14	37.50
DeS	Node 1	150	76.19	-5.88
	Node 2	122	59.32	23.96
	Node 3	55	45.00	15.38
	Node 4	43	38.89	4.17
Rijndael/AES	Node 1	38	37.50	20.91
	Node 2	105	42.86	33.00
	Node 3	46	64.71	-6.25
	Node 4	8	66.67	-38.10
Averages:		52.875	51.98	21.55

Table 2. Latency Reduction and Area Increase for DFGs with Template Generation

## 6. RELATED WORK

Instruction selection (in the context of code generation) is the fundamental technique used by a compiler to map its intermediate code representation to a target machine's set of operations. The best-known selection algorithm is an optimal dynamic programming solution, which was first devised by Aho and Johnson [9], extending the work of Sethi and Ullman on code generation for expression trees [10]. The goal of these works was to minimize the number of total program steps (or operations performed), especially those operations performed on registers. The target architectures of their machines were known in advanced and the generation of new templates was a lengthy and complicated procedure, requiring the interface of hardware and systems-software designers. Our work is a rethinking of the instruction selection process, attempting to create the hardware and its software simultaneously.

Template generation via clusters of primitive operations has been explored before in [11, 12, 13]. These techniques are used to identify regular sequential sets of operations for use in ASIP design or high-level synthesis. To the best of the authors' knowledge, this is the first work that utilizes slack calculations on dataflow graphs in order to select parallel clusters of nodes.

PipeRench (a fully reconfigurable, pipelined FPGA architecture) utilized regularity extraction to reduce circuit area and increase performance [14]. However, their templates were hand optimized to produce beneficial results. Cadambi and Goldstein limit the form of template they consider to single output templates with a bounded set of inputs, reasoning that inputs/outputs must be bounded in order for their macros to remain routable. However, in the architecture we discuss, these instructions are implemented as small hard-logic blocks integrated into the reconfigurable fabric. These blocks can be given additional routing resources. Therefore, the restrictions provided by the authors need not be considered in our work. The authors suggested that profiling may be beneficial for small granularity FPGAs, but no supporting experiment was provided in their work to support this suggestion.

Regularity extraction has many applications in the CAD domain, including hierarchical scheduling [15], reduction of data-path complexity and improved design quality [16], system level partitioning [17], and power reduction [18].

Rao and Kurdahi [17] discussed template generation for clustering (at the system level) using the first fit bin filling heuristic. Later, Cadambi and Goldstein [19] proposed single output template generation with a bottom-up approach. Both methods restrict templates by size and number of inputs/outputs. Our algorithms are flexible enough to support such a restriction (although currently we merely impose a size constraint).

IMEC's Cathedral Project [20] used a signal flow graph rather than a CDFG to perform clustering. However, their investigation was somewhat similar to ours in spirit. The Cathedral Project investigated DSP applications at the high-level synthesis stage. Their data path was composed of Abstract Building Blocks (ABBs) (instructions available from a given hardware library). A collection of many ABBs was referred to as an application specific unit (ASU). IMEC's algorithm attempted to identify ASUs that could be executed with the best performance via manual clustering of the graph into more compact operations (similar to our template construction). However, unlike IMEC's work, our template generation algorithms are automated. Their results showed that reduction of critical path length as well as reduction of interconnect were both achievable via this method.

One of the most motivating investigations of performance gain via template matching was demonstrated in Corazao et al [21]. Much like the traditional compiler template matching algorithms, their project assumed a given library of highly regular templates. These templates could substitute for the comprising operations during the high-level synthesis stage, leading to critical path minimization. If some parts of a template were not needed, these portions were allowed to go unused (leading to a partial template matching). Their experiment resulted in large performance gains with a small area increase. Although many optimization techniques were tried as part of their strategy, template selection was described as having the largest positive performance impact.

Compton and Hauck's Totem Project [22] seeks to automate the generation of reconfigurable architectures customized for a limited set of applications. During placement and routing, they map coarse-grained components to a one-dimensional data path axis. Whereas our algorithm is given an application as high-level specification, their input is a set of architecture netlists, which are transformed into a physical design while simultaneously targeting improved routing flexibility and decreased area.

## 7. CONCLUSION AND FUTURE WORK

In this work we have presented template generation as a solution to the problem of regularity extraction. In particular, we have extended previous work on template generation to include parallel templates, which offer both instruction level parallelism and a previously unobserved form of parallel regularity. We have introduced the All Pairs Common Slack Graph (APCSG) as a data structure on which parallel templates can be found, and have described and verified an algorithm to perform simultaneous sequential and parallel instruction generation. Our results demonstrate large performance gains on computationally intensive algorithms with minimal FPGA area increases.

In the future, we intend to parameterize our algorithm, enabling it to produce different instructions given different system constraints (such as maximum area increase). This will allow a new level of hardware awareness in our compiler, allowing it to improve latency only when it is physically reasonable to do so. Additionally, we wish to explore DFG inlining and translations to other intermediate representations, which will provide us with a more global view of the application, as well as help us to determine other forms of instruction regularity.

## 8. REFERENCES

- [1] R. Kastner, S. O. Memik, E. Bozorgzadeh, and M. Sarrafzadeh, "Instruction Generation for Hybrid Reconfigurable Systems," *Proceedings of the International Conference on Computer-Aided Design*, 2001.
- [2] P. Foggia, C. Sansone, M. Vento, "An Improved Algorithm for Matching Large Graphs," the 3rd IAPR-TC15 Workshop on Graph-based Representations, Ischia, 2001.
- [3] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion and M. S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler," *IEEE Computer*, December 1996.
- [4] Oja E. and Karhunen, J, "Recursive Construction of Karhunen-Loeve Expansions for Pattern Recognition Purposes," *Proc. of the 5th International Conference on Pattern Recognition*, Miami Beach, Florida, USA, Dec. 1-5, 1980, pp. 1215-1218.
- [5] Gallery of CSS Descramblers. <http://www-2.cs.cmu.edu/~dst/DeCSS/Gallery/>
- [6] National Bureau of Standards, NBS FIPS PUB 74, "Guidelines for Implementing and Using the NBS Data Encryption Standard," U.S. Department of Commerce, April 1981.
- [7] J. Daemen and V. Rijmen, "The Block Cipher Rijndael," *Smart Card Research and Applications*, LNCS 1820, J.-J. Quisquater and B. Schneier, Eds., Springer-Verlag, 2000, pp. 288-296.
- [8] S. O. Memik, E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh, "A Super-Scheduler for Embedded Reconfigurable Systems," *Proceedings of the International Conference on Computer-Aided Design*, 2001.
- [9] A.V. Aho and S.C. Johnson, "Optimal Code Generation for Expression Trees," *Journal of the ACM*, vol. 23, pp. 488-501, July 1976.
- [10] R. Sethi and J.D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," *Journal of the ACM*, vol. 17, pp. 715-728, October 1970.
- [11] J. Van Praet, G. Goossens, D. Lanneer, and H. De Man, "Instruction Set Definition and Instruction Selection for ASIPs," in *Proceedings of the 7th International Symposium on High-Level Synthesis*, pp. 10-16, May 1994.
- [12] C. Liem, T. May, P.G. Paulin, "Instruction-Set Matching and Selection for DSP and ASIP Code Generation," *Proceedings of the European Design & Test Conference*, pp. 31-37, February 1994.
- [13] O. Bringmann and W. Rosenstiel, "Cross-Level Hierarchical High-Level Synthesis," *Proceedings of the Design Automation and Test in Europe*, 1998.
- [14] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *Computer*, vol. 33, pp. 70-77, 2000.
- [15] L. Tai, D. Knapp, R. Miller, and D. MacMillen, "Scheduling Using Behavioral Templates," *Proceedings of the Design Automation Conference*, 1995.
- [16] T. J. Callahan, P. Chong, A. DeHon, and J. Wawrzyniec, "Fast Module Mapping and Placement for Datapaths in FPGAs," *Proceedings of the International Symposium on Field Programmable Gate Arrays*, 1998.
- [17] D. S. Rao and F. J. Kurdahi, "On Clustering for Maximal Regularity Extraction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, 1993.
- [18] R. Mehra and J. Rabaey, "Exploiting Regularity for Low-Power Design," *Proceedings of the International Conference on Computer-Aided Design*, 1996.
- [19] M. Kahrs, "Matching a Parts Library in a Silicon Compiler," *Proceedings of the International Conference on Computer-Aided Design*, 1986.
- [20] S. Note, W. Geurts, F. Catthoor, and H. De Man, "Cathedral-III: Architecture-Driven High-Level Synthesis for High Throughput DSP Applications," *Proceedings of the Design Automation Conference*, 1991.
- [21] M. R. Corazao, M. A. Khalaf, L. M. Guerra, M. Potkonjak, and J. M. Rabaey, "Performance Optimization Using Template Mapping for Datapath-Intensive High-Level Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, 1996.
- [22] K. Compton and S. Hauck, "Totem: Custom Reconfigurable Array Generation," *Proceedings of the Symposium on FPGAs for Custom Computing Machines Conference*, 2001.

## Automatic Detection of Recurring Operation Patterns

Marnix Arnold and Henk Corporaal  
Computer Architecture Laboratory  
Department of Electrical Engineering  
Delft University of Technology  
{marnix,heco}@cardit.et.tudelft.nl  
<http://cardit.et.tudelft.nl/MOVE/>

### Abstract

An important problem in the area of processor design for embedded systems is determining the proper instruction set architecture. Trade-offs have to be made between programmability and reusability of dedicated hardware for special functionality on the one hand, and a high performance dedicated instruction set on the other hand. This paper addresses the question of how to find specialized ISA extensions for a set of applications.

We describe the application of a new pattern matching technique to the problem of the identification of recurring patterns of operations. By implementing frequently occurring operation patterns in hardware, and using this hardware as special function units, a fine-grained hardware/software partitioning can be found. The fine granularity, and the fact that patterns are taken from a number of different target applications rather than a single one, increase the opportunities for reuse of the special-purpose hardware. We illustrate our technique with experiments on a number of benchmarks from the DSP domain.

**keywords:** pattern matching, co-design, design space exploration, instruction set synthesis.

### 1 Introduction

Hardware/software co-design is often performed on a per-application basis, yielding systems that are highly application-specific. The approach taken is usually a coarse-grained one: entire functions of the application are mapped either in hardware or software [4]. Any hardware thus generated is only reusable by other applications if those include the same function. For application-specific systems this is obviously not a prob-

lem. For application-*domain*-specific, reprogrammable systems, however, we may want to increase the granularity of the hardware/software partition, to increase the chances of hardware reuse. It is with this in mind that we consider a partitioning approach that centers on groups of instructions rather than entire functions. In this paper, we present a new algorithm for the automatic, on-the-fly detection of groups (patterns) of instructions as they occur in the application(s) that we want to generate an execution engine for. Patterns that occur frequently among the target applications can then be considered for implementation in hardware.

Section 2 discusses work from the related areas of instruction set synthesis, technology mapping and code generation. An overview of the algorithm used for the detection of recurring operation patterns is given in section 3. Experiments and their results are described in section 4. We conclude this paper with section 5.

### 2 Related Work

Pattern matching techniques have been around for quite some time, originating from the string matching problem [1]. Keutzer [5] first applied pattern matching techniques to the problem of technology mapping, noting similarities with the code generation problem [2]. Recent work by Kukimoto [6] extended these techniques to allow for rooted-DAG-shaped subject graphs, as opposed to tree-shaped graphs. Liao et.al. [7] use a binate covering technique that allows the subject graph to have any geometry. However, no matching techniques are available that deal with *patterns* that are not trees or single-output DAGs [9]. We would like to detect and exploit such patterns, though, so we were forced to come up with a new matching algorithm [3].

The work in this paper is somewhat related to the field of instruction set synthesis. Liem et.al. [8] use matching and covering techniques to identify recurring instances of patterns from a predefined library, rather than constructing this library automatically, as we will do. The search for new operation patterns as described

in [10] is limited to chains (sequences) of operations, whereas we consider patterns of any shape.

### 3 Algorithm Overview

For the matching of pattern graphs that are not trees, conventional techniques are not suitable. For this reason, we have come up with a new matching algorithm, based on the principle of *incremental matching*. In this section we will give a brief overview of the matching algorithm and how it can be extended to automatically construct a library of recurring patterns. The covering algorithm we employ in section 4 will not be discussed, for the purposes of this paper it will suffice to say that it is a heuristic method, based around the dynamic programming approach taken in tree covering. A comprehensive description of the algorithms can be found in [3].

#### 3.1 Incremental Matching

Given a subject graph  $G_{sub}$  and a library of pattern graphs  $PatLib = \{G_{pat_i}\}$ , as shown in figure 1, we iterate over all operation nodes in  $G_{sub}$  (nodes I, II and III), in no particular order. Each of the subject graph's operation nodes  $N_{sub}$  is matched against the pattern graphs' operation nodes  $N_{pat}$  that have the same opcode. Each time a pair of operation nodes ( $N_{sub}, N_{pat}$ ) meets certain matching criteria, which we will not go into here, a *partial match* is created. In figure 2, the partial matches  $m_1$  and  $m_2$  constructed for pattern graph  $G_{pat4}$  are shown (there would, of course, also be partial matches for the other patterns). As can be seen, a match is really nothing more than two reference vectors that refer to subject operations and operands, respectively. The position a reference occupies in a reference vector indicates with which pattern operation (operand) it corresponds to.

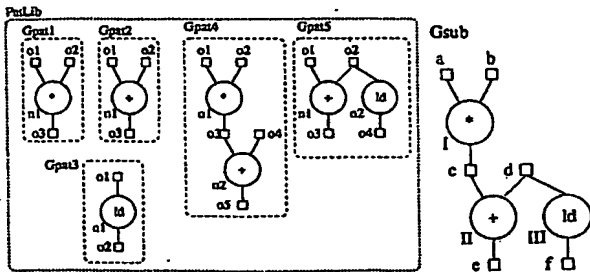


Figure 1: Example of pattern and subject graphs.

Looking at the partial matches shown in figure 2, we notice that the matches  $m_1$  and  $m_2$  share the reference to subject operand node c. These two matches can be

combined into a new match  $m_3$ , which is the union of the two. This match has no empty spaces in its reference vectors and is said to be *complete*. In the same manner, we can combine partial matches for other patterns into complete matches, regardless of the shape of those patterns. Indeed, pattern graph  $G_{pat5}$ , which is a multiple-output DAG, is matched in the exact same way as graph  $G_{pat4}$  from our example.

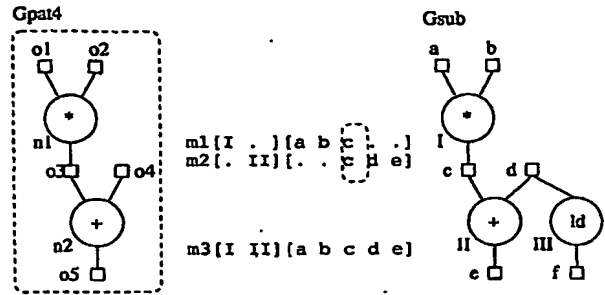


Figure 2: Partial matches  $m_1$  and  $m_2$  merge into the complete match  $m_3$ .

#### 3.2 Library Construction

Another drawback of using a conventional matching algorithm is that it requires the pattern library to be defined beforehand. This implies that we need some advance knowledge of which patterns to expect, but such knowledge may not be available. We will show that this problem can be overcome by extending the incremental matching algorithm to include automatic library construction capabilities.

As before, we start with a subject graph  $G_{sub}$  and a library of pattern graphs  $PatLib = \{G_{pat_i}\}$ , as shown in figure 3. The difference with the previous example is that the initial pattern library now only contains a minimal set of patterns: just the ones with a single operation node. When we start processing the subject graph's operation nodes, we will initially only construct matches for the single-operation pattern graphs. Whenever we finish constructing partial matches on an operation node, we can now look for opportunities to create new pattern graphs.

In the example of figure 3, after we finish processing operation nodes I and II (again in an arbitrary order), we see that two matches  $m_1$  and  $m_2$  both contain a reference to operand node c. Note that now the matches refer to different patterns, which was not the case in the example of figure 2. By combining the matches  $m_1$  and  $m_2$  into a new match  $m_3$ , we have a recipe for constructing a new pattern! By copying all the nodes (operations and operands) referenced in  $m_3$  into a new

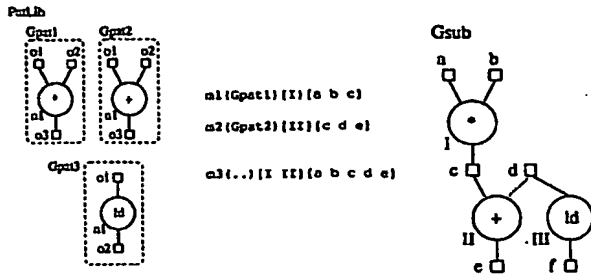


Figure 3: Pattern construction by combining matches.

pattern and adding it to the library, we will be able to log all future occurrences of this pattern.

#### 4 Experiments

We execute our method for detecting recurring patterns of operations on a number of well-known benchmarks from the DSP domain. In this paper, we limit the size of the patterns to two operation nodes, even though the pattern detection and matching algorithms can handle patterns of arbitrary size. An overview of the benchmarks, with their dynamic operation counts, is given in table 1.

Name	Description	#ops
bspline	FIR Filter	6149
compress	Compression (dct 2d)	163513
dft	Discrete FFT	6666
edge	Edge detection	268717
expand	Decompression (idct 2d)	151083
feowf	5th Order Elliptic Wave	13067
fir	35 pt. Lowpass FIR	30459
flatten	Level histogram of image	33960
iir	IIR highpass filter	10794
pse	Sehwa's FIR filter	6917
smooth	Convolution w. 3x3 kernel	83365

Table 1: DSP benchmarks.

Each benchmark is trace simulated, and the pattern detection experiments are performed on the dataflow graph of the execution trace. The detected patterns for all benchmarks are then put into a unified pattern library. This library is then used to cover the execution trace of each benchmark, to get a feeling for how much each pattern would help reduce the operation count of that benchmark. These coverings can be seen as the best results the covering algorithm can attain with an unbounded pattern library (after all, all patterns ever detected in any of the benchmarks are there).

The reason we use an execution trace rather than the static object code for our experiments is that a trace effectively masks control flow, making pattern matches visible that reach across basic block or even loop boundaries. If we cover the trace with the patterns we found, then the matches that cross control flow boundaries would seem to indicate that code motion (speculative execution, loop unrolling, etc.) could be beneficial.

**Constructing a unified library** After covering, we calculate how often each pattern was used for each benchmark. Note that it is misleading to just count the matches for that pattern, as it is unknown how many of those (likely to be mutually exclusive) matches will be chosen for the cover. The patterns can now be sorted according to how much they contributed to the unbounded-library covering (i.e. as a percentage of the total number of matches that were chosen for the cover). Since we are most interested in patterns that contribute to the operation count reduction of the entire application domain, rather than just one application, we sort the patterns by the average of their contributions to the per-benchmark coverings. The reason we average the contributions of the patterns (a percentage) rather than the absolute share in the coverings (a number of pattern instantiations) is that we do not want to skew our results towards the larger benchmarks. This yields a unified, sorted pattern library in which all benchmarks are represented equally.

**Unbounded library covering analysis** In figure 4, we see the (cumulative) contribution to the individual benchmarks' coverings of pattern libraries that consist of the top- $x$  patterns of the unified library. As can be seen, some benchmarks get a better-than-average contribution (bspline, pse) and some get a worse-than-average contribution (foewf). This can be interpreted as follows: the more an application's contribution graph pulls towards the upper-left corner of the figure, the more 'average' or representative for the application domain the application is. As a consequence of this, figure 4 can also be used to judge how well the applications fit together on the same instruction set, something which is difficult to determine by inspection of the benchmarks' source code alone. A final remark on figure 4: it can be seen that from the top-80 patterns onward, there is no additional contribution to the covering of any of the benchmarks (the 100% mark has been reached). This implies that none of the patterns added to the library from that point onward are ever actually used in the coverings of any of the benchmarks.

**Covering with partial libraries** Now that we have found a ranking of patterns in the unified library, we can check if there is a relation between the contribution

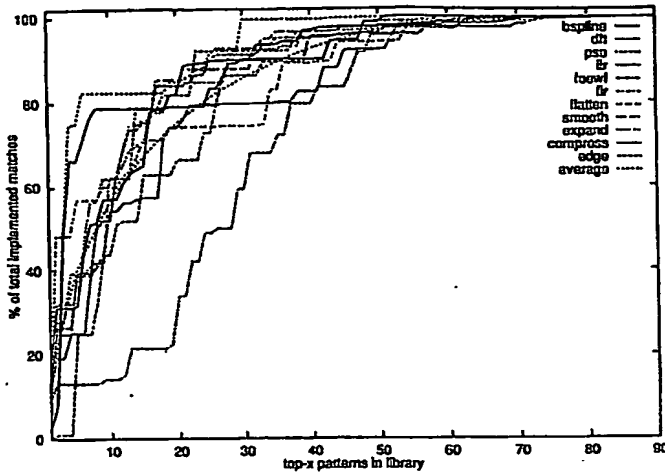


Figure 4: The contribution of patterns to the coverings.

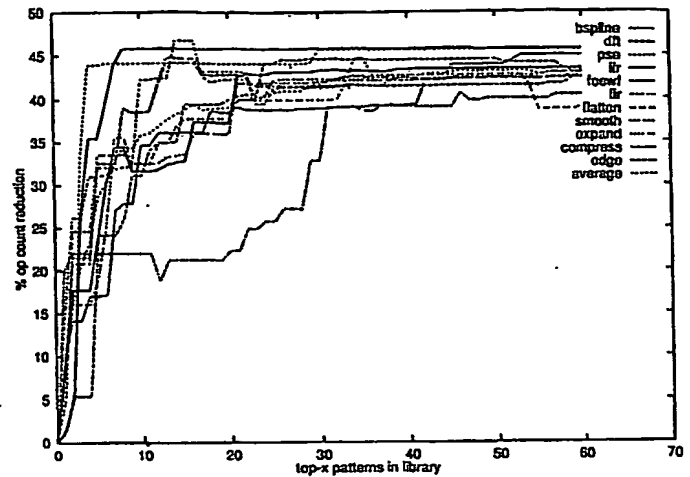


Figure 5: Operation count reduction for the incremental pattern libraries.

of the patterns in the top- $x$  libraries to the 'ideal' (unbounded library) covering, and the actual results of the covering algorithm for each of the top- $x$  libraries. For this, it is necessary to re-cover each of the benchmarks using each of the top- $x$  pattern libraries. The results of these coverings can be found in figure 5. It can be expected that if a library has a lower-than-average contribution (in figure 4); then the operation count reduction will also be below average. This is confirmed by the curve for the foveal benchmark. Similarly, benchmarks with a higher-than-average contribution should have a higher-than-average operation count reduction. The operation count reduction curves for the bspline and pss benchmarks confirm this.

Note that the curves in figure 5 do not increase monotonously, which is what we would expect if we give the covering algorithm an extra pattern to work with each time. These occasional dips are due to the fact that the covering algorithm is a heuristic method, which every once in a while gets confused and yields a sub-optimal result. Also note that 50% operation count reduction is a hard limit: since, for the purposes of this experiment, we only use patterns that are no larger than two operations, the best result we can theoretically get is obtained if all operation are replaced, in pairs, with two-operation patterns. In addition, it must be noted here that, since the covering algorithm operates on execution traces and hence ignores control flow, the operation count reduction figures must be seen as upper bounds for the operation count reduction that a compiler can achieve when performing code generation (when control flow is taken into account).

**Analysis of the top 10 patterns** The top ten patterns of the unified library are shown in figure 6. The most popular pattern (nr.1) is an integer add, followed by another integer add. In hardware, this can be implemented as an add, followed by an accumulate on the same unit, or as a 3-input, 2-output unit that can execute the pattern in a single processor cycle [11]. The second pattern is a conditional jump, where the condition is calculated by the greater-than node. Patterns 5 and 7 are array references, where the address calculation consists of a base-plus-offset calculation. Pattern 6 is the well-known multiply-add, pattern 8 the almost equally well-known add-shift. Note that pattern 9 performs the same calculation on both nodes! It looks as if the compiler missed an optimization opportunity, possibly hidden by control flow.

It can be seen that the top 10 patterns are not proper trees, since most also export their intermediate values. However, with the exception of pattern 9, none of the patterns represent operations that execute in parallel. It is quite possible that this is an artifact of the covering heuristic, which favors chains of operations. It will be interesting to see whether this situation changes if we come up with a different covering strategy.

## 5 Conclusion and Future Work

We presented a technique for identifying common operation patterns across a range of applications, using a new pattern matching algorithm. This algorithm is innovative in that it can handle patterns of arbitrary shape, widening the scope of the search for operation patterns that can be implemented in hardware. Newly

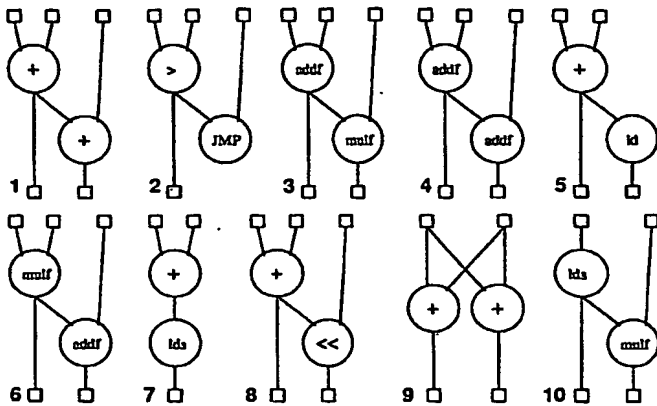


Figure 6: The top 10 patterns.

discovered patterns are added to the pattern library on-the-fly, resulting in a single pattern detection (finding new patterns) and matching (marking occurrences of patterns) pass.

Using the new technique, we found patterns of operations common to a set of benchmarks, which, when covering is applied, indicate that a substantial operation count reduction is possible (e.g. 20 patterns yield an average operation count reduction of 40%). Furthermore, we were able to incrementally construct a library of new operations (patterns) and analyze the influence of each new operation on the average operation count as well as on the operation count of each benchmark separately.

The covering heuristic, which was based on dynamic programming, still leaves something to be desired. The influence of the covering algorithm on the selection of the patterns of various shapes is not well understood at this point. Different algorithms may yield different top-x pattern libraries, which is something that needs to be investigated.

It has already been noted that our method operates on (dynamic) execution traces rather than (static) code. The absence of control flow in an execution trace can be seen as both an advantage (patterns invisible in static code can be detected) and a disadvantage (covering results are upper bounds, rather than actual, attainable values). In the future, we will concentrate on how to apply our current techniques to the problem of code generation.

## References

- [1] A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic research. *Communications of the ACM*, 18(6):333-340, June 1975.
- [2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.
- [3] Marnix Arnold and Henk Corporaal. Matching and covering with multiple-output patterns. Technical Report 1-68340-44(1999)-01, Delft University of Technology, <http://cardit.et.tudelft.nl/MOVE/papers/Arnold99a.ps>, 1999.
- [4] Peter M. Athanas and Harvey S. Silverman. Processor reconfiguration through instruction set metamorphosis. *IEEE Computer*, (0018-9162/93/0300-0011):11-18, 1993.
- [5] K. Keutzer. Dagon: Technology binding and local optimization by dag matching. In *DAC, Proceedings of the Design Automation Conference*, pages 617-623, May 1987.
- [6] Yuji Kukimoto, Robert K. Brayton, and Prashant Sawkar. Delay-optimal technology mapping by dag covering. In *Proceedings of the Design Automation Conference*, 1998.
- [7] Stan Liao, Srinivas Devadas, Kurt Keutzer, and Steve Tjiang. Instruction selection using binade covering for code size optimization. In *Proceedings of 1995 International Conference on Computer-Aided Design*, pages 393-399, 1995.
- [8] Clifford Liem, Trevor May, and Pierre Paulin. Instruction-set matching and selection for dsp and asip code generation. In *Proceedings of EDAC-ETC-EUROASIC*, pages 31-37, 1994.
- [9] Giovanni De Micheli. private communication, 1998.
- [10] Frederick Onion, Alexandru Nicolau, and Nikil Dutt. Compiler Feedback in ASIP Design. Technical report, University of California, Irvine, September 1994.
- [11] Stamatis Vassiliadis, James Phillips, and Bart Blanner. Interlock Collapsing ALU's. *IEEE Transactions on Computers*, 42:825-839, July 1993.

# Matching and Covering with Multiple-Output Patterns

Marnix Arnold  
Delft University of Technology  
Department of Electrical Engineering  
Section Computer Architecture and Digital Technique  
*marnix@cardit.et.tudelft.nl*

January 14, 1999

Technical Report 1-68340-44(1999)-01  
DRAFT VERSION

## 1 Introduction

The algorithms and data structures described in this report are aimed at the automatic detection of patterns of instructions in a (dynamic) execution trace of a program. Two separate phases can be distinguished: a pattern library construction phase, and a matching and covering phase.

During the library construction phase, all patterns up to a certain size are found and their occurrence frequencies are noted. The first part of the matching and covering phase consists of a matching sub-phase, in which all occurrences in the execution trace of all patterns from a given pattern library are found. This library can be user-specified, but the patterns found during the pattern library construction phase can also be used. In the second part, the covering sub-phase, a selection is made from the matched patterns in such a way that the data-ready time of the slowest trace output is minimized when the selection (cover) is implemented.

This report describes the algorithms used in the matching and covering phase. The pattern library construction phase uses a modified version of the matching algorithm and will be described last.

In the next section, an overview of related work is given. The strategy for library construction, matching and covering using partial matches is explained by means of an example in section 3. In section 4 a description of the data structures used in the implementation of partial matching and covering can be found. Sections 5 and 6 contain in-depth descriptions of the matching and covering algorithms, respectively. The library construction algorithm is discussed as part of the matching algorithm in section 5.5. Conclusions are stated in section 7.

## 2 Related Work

Matching and covering algorithms are well-known in the fields of code generation and logic synthesis. Keutzer [1] was the first to recognize the similarity between the software compiler's task of generating code and the technology mapping problem in automated VLSI design. Both problems can be handled with a matching algorithm,

to find all possible instantiations of patterns (instructions or standard cells), followed by a covering algorithm to make a selection of matches that optimizes some criterion (code size, VLSI area or latency, etc.).

As described in [3], there are two main approaches to handling the matching problem when performing technology mapping: the Boolean and the structural approach.

The Boolean approach can only be applied to networks of Boolean functions, since it operates by checking the equivalence of functional representations of the patterns and functions representing portions of the network. This equivalence is detected using (Ordered) Binary Decision Diagrams, which makes this technique unsuitable for networks of non-Boolean functions.

Structural matching will work on networks containing nodes of any type of function. Since this approach focuses on the identification of common patterns, the subject and pattern graphs have to be written in terms of the same types of function nodes. Most approaches ([1], [4]) require all graphs to be trees (single-output, acyclic, non-reconvergent graphs). It is therefore necessary to decompose the subject graph into a set of disjoint trees. For solving this decomposition problem only heuristical methods exist.

In a more recent work by Kukimoto et al. [2], a structural matching method was introduced that can handle DAG-shaped subject graphs, allowing reconvergence within the graph. However, patterns are still restricted to single output, tree-shaped graphs. When this method is used for technology mapping, rather than tree-mapping, faster solutions are found (up to 67% faster), at the cost of a significant increase of area (up to 126% larger). This area increase is due to the fact that the DAG mapping approach freely duplicates subject nodes whenever a pattern match covers a multiple-fanout point. This may not be a problem in VLSI synthesis, where extra cells are cheap in terms of chip area (although it may be desirable to have some control over the node duplication aggressiveness). In code generation, however, node duplication leads to extra instructions. On a machine with limited execution resources this may cause the schedule length to increase and hurt the execution speed of the code rather than helping it.

None of the previously described matching algorithms allow pattern graphs to have more than one output. Our proposed matching algorithm does not have this restriction, making it possible to exploit a larger family of pattern graphs.

### 3 Introducing the matching and covering strategy

Given a subject graph  $G_{sub}$ , consisting of the operations and operands in an execution trace, we must try to find all matches between pattern graphs  $G_{pat}$  (from a pattern library *PatLib*) and subgraphs of  $G_{sub}$ . The strategy for this is described informally in section 3.1.

The search for new patterns is remarkably similar to the detection of matches with existing patterns. An extended version of the pattern detection strategy can be used, as described in section 3.2.

After all matches are found, we try to find the best *cover*, or selection of matches, that is, the set of matches that, when implemented, minimizes the data-ready time of the longest path through the subject graph. The covering approach is described informally in section 3.3.

#### 3.1 Matching

In figure 1, the subject graph and some pattern graphs are shown. Note that it is generally necessary to have patterns for all individual operations, to ensure that

there is always at least one match to fall back on for each subject graph operation node, during covering.

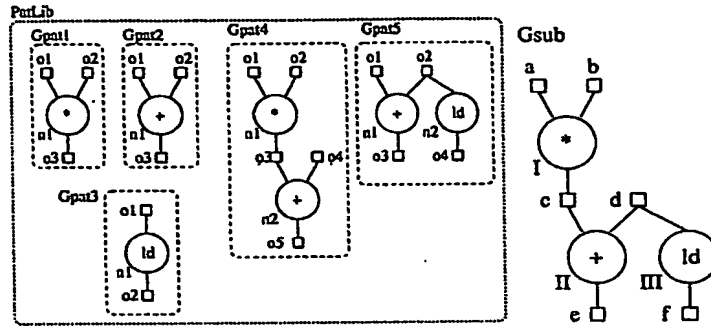


Figure 1: Example of pattern and subject graphs.

From the example it can be seen that several nodes from pattern graphs of *PatLib* correspond to nodes in  $G_{sub}$ . These relationships are called *partial matches*. A match always pertains to a single pattern graph, and it contains a vector of references to subject nodes. The position of a reference within the vector indicates which pattern node it corresponds to. Figure 2 illustrates partial matches and a match on a pattern. Since operation node *I* of  $G_{sub}$  matches operation node  $n_1$  of  $G_{pat1}$ , a partial match  $m_1$  can be created for node *I* and its operands. This match consists of a vector of references to subject graph nodes, corresponding to pattern graph nodes that are being matched. In the figure,  $m_1$  is a partial match for  $G_{pat1}$  with references to subject operation node *I* (at the position corresponding to  $n_1$ ) and subject operand nodes *a* (for  $o_1$ ), *b* (for  $o_2$ ) and *c* (for  $o_3$ ). The rest of the entries in the match vector remain empty. In a similar fashion, partial match  $m_2$  can be constructed for subject operation node *II* and its operands.

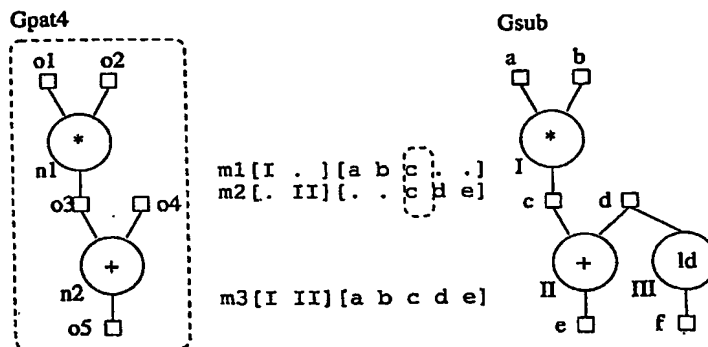


Figure 2: Partial matches merged into a complete match.

In order to get *full matches* from the partial matches that can thus be found, it is necessary to combine partial matches where possible. In our example, it can be seen that combining partial matches  $m_1$  and  $m_2$  yields a full match for  $G_{pat1}$ . It is obviously not correct to combine just any two matches from anywhere in the subject graph. There are several conditions under which it is allowed, however.

These will be discussed in detail in section 5. For the moment we will say that the two matches in the example may be combined because there is one vector entry the same for both matches (the third entry of both  $m_1$  and  $m_2$  contains a reference to operand node c). The resulting match  $m_3$  is a full match for  $G_{pat1}$ .

After all valid combinations of partial matches have been made, all matches that are not full matches (i.e. matches with blank entries in their reference vector) can be deleted. The resulting set of matches is used to find a cover for the subject graph.

### 3.2 Automatic pattern library construction

In figure 1, the pattern library was introduced. It contains a number of pattern graphs, for which the matching algorithm must find matches with the subject graph. All patterns in the library were put there beforehand, which implies some advance knowledge of which patterns are expected to occur in the trace. If no such knowledge is available, however, it is useful to have a mechanism that automatically discovers which patterns occur most frequently. It is with this in mind that the pattern library construction phase is introduced.

Starting with a library that only contains basic operation patterns (patterns  $G_{pat1}$ ,  $G_{pat2}$  and  $G_{pat3}$  in figure 1), we construct matches for the first subject operation node (node I) in the same way as described in section 3.1. This yields a match  $m_1$  for  $G_{pat1}$ , as shown in figure 3. The same is done for node II, yielding match  $m_2$ . There are now two matches on operand node c and we will attempt to merge them into a new pattern. This is done by taking the union of the reference vectors of  $m_1$  and  $m_2$ , resulting in the new match  $m_3$ .

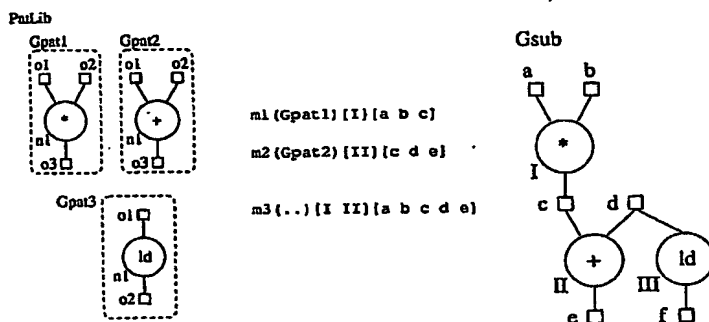


Figure 3: Pattern construction by taking the union of matches.

We can now construct a new pattern graph by copying the subject operations and operands referenced in  $m_3$ . This results in a graph  $G_{pat4}$  which is equivalent to the one also called  $G_{pat4}$  in figure 1. Before we do this, however, we must ensure that the pattern we are about to construct is not already contained in the pattern library. Rather than checking all patterns in the library against the one to be constructed (a lot of work if the library contains many patterns), we check the matches on the current operand node (d) against the newly formed  $m_3$ . If the library contains any pattern which is equivalent to the new one, then that pattern will have been matched with the exact same subject nodes, i.e. a match will have been formed on d that references all the same nodes as  $m_3$ . It is therefore sufficient to check for the existence of a complete match on d that has the same reference sets as  $m_3$ . If there is none, we can add the new pattern graph to the library.

Now that a new pattern has been added to the library, we will detect all subsequent matches for that pattern automatically. In this way, all possible combinations of operation and operand nodes can be found and kept track of as they occur in the subject graph. To prevent the number of patterns from growing unacceptably large, some limit will have to be imposed on both the number and size of the pattern graph, but theoretically any number of patterns of any size can be found.

### 3.3 Covering

Given a subject graph with full matches found for all nodes, we now try to find the subset of the full matches that, when implemented, minimizes the data-ready time of the slowest subject graph output. This process is called *covering*.

Until now, we have only looked at patterns as a single graph of operand and operation nodes, with the operation set of the latter being the same as that of the subject graph. However, each of these patterns, or *source graphs*, corresponds to a faster, cheaper or more efficient implementation of the same functionality. This corresponding implementation is also represented in a graph, the so-called *destination graph*. It contains the operand and operation nodes that have to be substituted into the subject graph for the nodes matching the pattern. Source and destination graphs are described in more detail in section 4.1.

In real life, a multiply operation usually takes two cycles and an addition one. The multiply-add operation however (corresponding functionally to pattern  $G_{pat4}$ ) usually only takes two cycles instead of three. Figure 4 shows the pattern library of the example again, with a latency matrix for the destination graphs. The single-node patterns are their own destination graphs, they are only in the library so that there is always a pattern to fall back on if no complex patterns matched for a certain node.

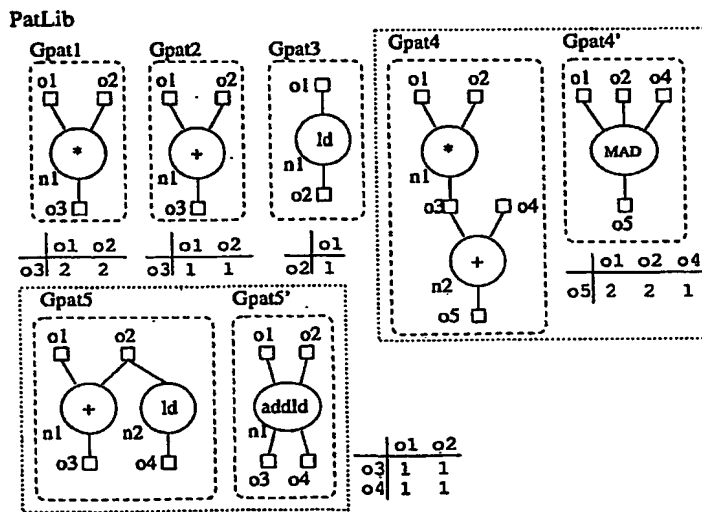


Figure 4: Pattern library with latency information.

Now that we have the latencies for the patterns, we can calculate the data-ready times that implementing the various matches would yield. Figure 5 shows the subject graph again, with all full matches that have been found in the matching phase.

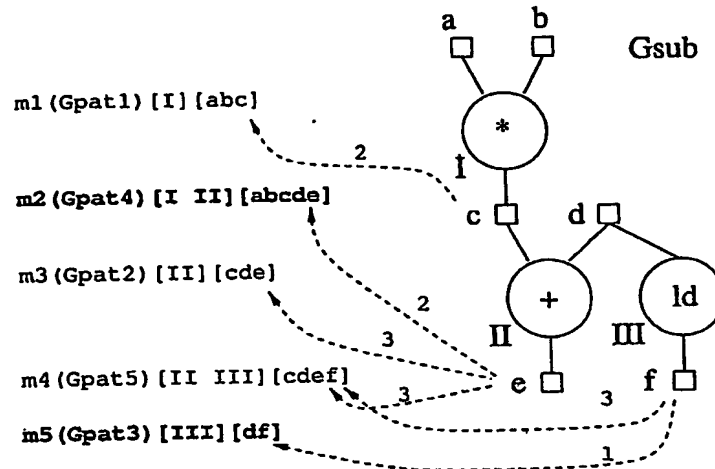


Figure 5: Subject graph with matches.

The first thing we need to do is determine which matches are so-called *output matches* for which operand nodes. A match is an output match for an operand node if that operand corresponds to an output of the pattern that is matched. In our example,  $m_4$  is an output match of nodes  $e$  and  $f$ , but not of  $c$  and  $d$ . In figure 5, pointers are shown from each operand node to all its output matches.

The next step is to determine the ready time of each operand for each of its output matches. For now, we assume a ready time of 0 for each of the subject graph inputs  $a$ ,  $b$  and  $d$ . The ready time of the other operand nodes is determined by the latency through the match under consideration and the best (lowest) possible ready time of its inputs, which is in turn determined by the best applicable match there. In our example, the ready time on node  $e$  is 2 for  $m_2$  (the maximum of the latency from any input of pattern  $G_{pat4}$  to output  $o_4$ , added to the ready time of that input, in this case 0). The ready time for  $m_3$  is the latency through pattern  $G_{pat2}$  (from  $o_1$  to  $o_3$ ) added to the ready time of its input  $o_1$  (corresponding to node  $c$ ). The data-ready time of  $c$  in turn is found by taking the lowest possible value for all the matches found there (in this case there is only one match,  $m_1$ ) which yields a ready time of 2. This results in a ready time of 3 on  $e$  for  $m_3$ . The algorithm to determine all ready times for all matches on all operands is discussed in-depth in section 6. In figure 5, the latencies for all output matches on all operand nodes are annotated on the dashed edges.

Now that we have a measure of quality for all matches on each operand node (the latency that they would attain), we can sort them by order of preference. For example, node  $e$  would have match  $m_2$  as favourite, with  $m_3$  and  $m_4$  in second and third place. We can also say which subject graph output we want to give precedence when we start covering, by looking at its best possible ready time. In our example,  $e$  would have to be covered first (best latency is 2), followed by  $f$  (best latency 1)<sup>1</sup>.

With the matches on the subject graph outputs sorted, we can now begin the actual covering. Starting at  $e$ , we choose the match that causes the lowest latency,

<sup>1</sup>Note that this is a heuristic 'best guess' and in no way a guarantee for an optimal solution, since it is very well possible that implementing a match on one path invalidates the favourite match on another path. For example, suppose  $m_4$  would have been the match of choice for node III, but we would implement  $m_2$  first, then  $m_4$  would no longer be valid unless we resort to node duplication, which introduces a whole set of problems of its own that we will not discuss here.

in this case  $m_2$  (latency 2). All matches that overlap  $m_2$  have to be invalidated ( $m_1, m_3, m_4$ ) since they can no longer be implemented (not that we would want to in this case). Now we proceed to cover all operands that are inputs to match  $m_2$ , but these are subject graph inputs  $a, b$  and  $d$  so we have finished covering along this path. Now we start covering on the last subject graph output:  $f$ . The best available match is  $m_5$ , which is still available, so we can implement it. We have now finished constructing a cover for the subject graph, consisting of matches  $m_2$  and  $m_5$ .

## 4 Data Structures

After the general description of the workings of the matching and covering algorithms in the previous section, we can now determine what the data structures representing the various data entities should look like. They fall into three main categories: the pattern library, the subject graph and the matches. Each of these is described in the following subsections.

### 4.1 The pattern library

For the description of the pattern library data structures, we shall begin by describing the most basic elements first, then construct more complex elements out of these. There are several levels of hierarchy in the pattern library structure, each will be described in the following paragraphs.

**Pattern graph nodes** At the most basic level, there are data structures to describe pattern operation and operand nodes. These nodes are interconnected and reference each other: operation nodes refer to operand nodes as operands and results, and operand nodes refer to operation nodes as definers and users. Figure 6 shows the relation between the data structures graphically. Not shown is the type of every operand: integer, floating-point or boolean variable, or immediate, in which case its value is also stored. Also not shown are references to the parent graph of every operation node.

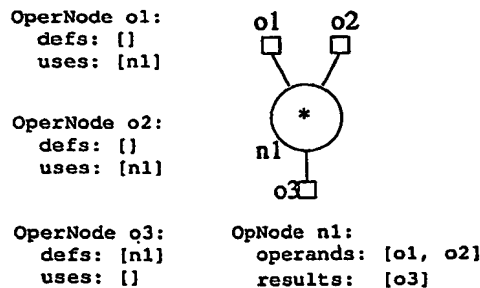


Figure 6: Pattern graph node data structures.

**Pattern graphs** Pattern graphs are built up out of pattern operation and operand nodes, which were described in the previous paragraph. Two types of pattern graphs can be distinguished: pattern source graphs and pattern destination graphs. Source graphs are used to match the subject graph against, they therefore only contain

operations from the same operation set as the subject graph. When a match is implemented, the destination graph corresponding to the source graph is substituted into the subject graph for the matching subject nodes. Destination graphs contain operations from the extended operation set (including special functionality).

Figure 7 shows the relation between pattern source and destination graphs and their components. Not shown are the latency figures from each input to each output; these can be calculated from the latencies of the individual operations in the graphs. For efficiency reasons it makes sense to calculate all latencies ones and store them in the pattern graph data structure.

Note that the names of input and output operands for corresponding source and destination graphs must be the same, otherwise it is not clear how the substitution should be performed. Also note that it is possible for several source graphs to correspond to the same destination graph, so that commutativity can be taken into account. In figure 7, another valid source pattern would be the one where operation  $n_2$  would have  $o_4$  as its first operand and  $o_3$  as its second.

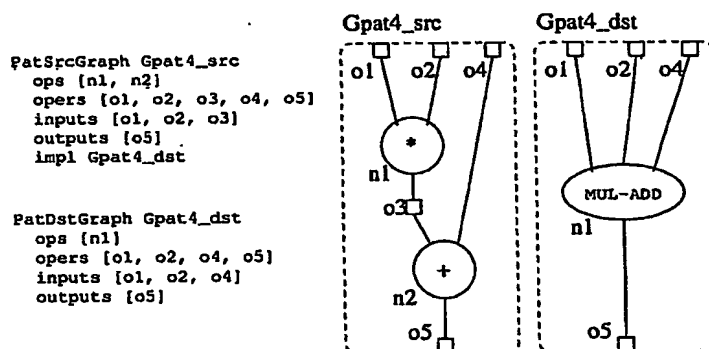


Figure 7: Pattern graph data structures.

**Pattern library** The pattern library is a set of pattern source graphs, for use in matching, and a set of pattern destination graphs, that are substituted during covering for the subject nodes that match the corresponding pattern source graph.

## 4.2 The subject graph

As was the case for the pattern graph data structures, the subject graph data structures will be described from the bottom up. Each level of hierarchy will be described in the following paragraphs.

**Subject graph nodes** At the lowest level of the hierarchy are the subject graph operation and operand nodes. They are roughly the same as the pattern graph operation and operand nodes, so there is no need for a detailed description here. The main difference is that the subject graph operand nodes have a list of references to matches (see section 4.3).

**Subject graph** The subject graph is a network of the subject graph nodes described in the previous paragraph. It represents a window in a dynamic execution trace of a program.

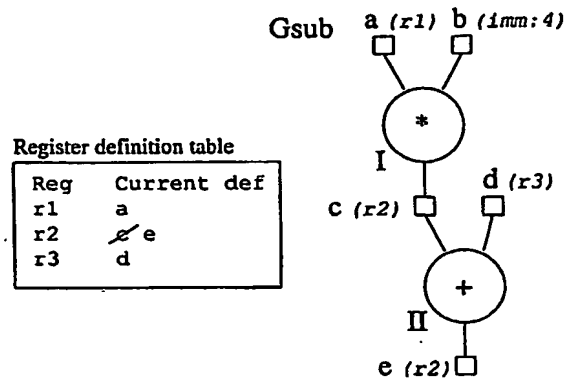


Figure 8: The register definition table.

**Register definition table** A special data structure has to be introduced to cope with the fact that we are dealing with a dynamic execution trace: the register definition table. During the trace simulation, each time a register is written, the corresponding register definition table entry keeps track of which operand node definition that write corresponds to. The previous definition of the register (operand node) can now no longer get additional uses. This information is necessary in order to determine partial matches, as will be described in section 5.1.

An example of the workings of the register definition table is shown in figure 8. After operation II has been executed, the current definition of register  $r_2$  is operand node  $e$ . The previous definition,  $c$ , can no longer be used by subsequent operations in the trace simulation since its value has been overwritten.

### 4.3 The match

The match is the central data structure for the matching and covering algorithms. It contains a reference to the pattern source graph that it is a match for. A vector of references to subject operand nodes indicate which subject nodes match which pattern node (from the aforementioned pattern source graph). The reference vector is ordered according to the ordering of the nodes in the pattern graph, so there is a direct correspondence between pattern nodes and subject graph nodes. If all positions in the reference vector are filled, the match is said to be complete.

We define the *degree* of a match as the number of operation nodes that are encapsulated by it. For example, if all input and output operands of two operation nodes are referenced in the match, the match is said to be of degree 2. If the operations themselves would be referenced in the match, we could simply count them and arrive at the same degree. If a match is complete, its degree is the same as the number of operation nodes in the corresponding pattern graph.

The subject graph operand nodes that are referenced by a match also have a reciprocal reference, paired with an index indicating the position the operand node occupies in the match's reference vector. Figure 9 shows an example of the relations between pattern and subject nodes and the match. On subject operand node  $a$ , there is a reference to match  $m_1$ , index 1. Match  $m_1$  references pattern  $Gpat4$ , so  $a$  corresponds to the pattern operand node at position 1 of  $Gpat4$ , which is  $o_1$ .

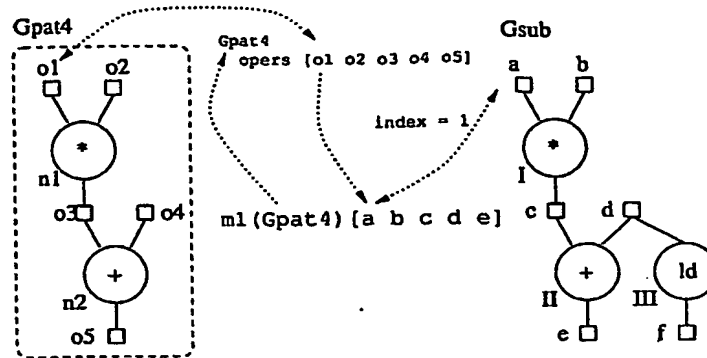


Figure 9: Relation between pattern and subject nodes and a match.

## 5 Matching

This section contains an in-depth description of the matching algorithm that was introduced in section 3. The algorithm makes use of the data structures outlined in section 4. First an enumeration of the basic assumptions will be given.

### Given

- A subject graph  $G_{sub}$  made up of subject operation nodes  $N_{sub}$  and subject operand nodes  $O_{sub}$ , interconnected by def- and use-edges.
- A library  $PatLib$  of pattern graphs  $G_{pat}$ , consisting of pattern operation nodes  $N_{pat}$  and pattern operand nodes  $O_{pat}$ , interconnected by def- and use-edges.

The matching algorithm consists of three phases:

1. **Partial match finding:** a first degree (i.e. matching only one operation of the pattern graph) partial match construction phase.
2. **Merging matches:** the partial matches found in the previous phase are merged into higher degree partial matches and complete matches.
3. **Cleanup:** all non-complete matches are removed.

### Matching

```
{
  FindPartialMatches();
  MergeMatches();
  CleanupMatches();
}
```

The three phases are described in the following subsections. A variation, integrating the first two phases into one, is described in subsection 5.4. Section 5.5 contains the description of an extension of the matching algorithm that detects patterns and stores them in the pattern library.

## 5.1 Partial match finding

In the partial match finding phase, we determine for each subject graph operation node which pattern graph operation nodes it corresponds to. We only look at the nodes themselves (and their operands), not their preceding and succeeding operation node context. This potentially leads to a lot of partial matches that can never become full matches, but it reduces the computational complexity of the algorithm. A tradeoff between computational complexity and number of matches created can be made by adding criteria to the *Nodematch* function, as explained below.

**FindPartialMatches** At the top level of the partial match finding algorithm, there are three nested loops. The outer loop iterates over all the subject operation nodes  $N_{sub}$  of the subject graph  $G_{sub}$ , the loop nested within iterates over all the pattern graphs  $G_{pat}$  in the pattern library *Patlib*, and the inner loop iterates over all pattern operation nodes  $N_{pat}$  of  $G_{pat}$ . Note that any of the loops may iterate over their domain in any particular order; one of the properties of the partial match finding algorithm is that topological order is irrelevant. Each pair of nodes ( $N_{sub}$ ,  $N_{pat}$ ) is compared, and if they match, a new *Match* is created for the pair and their respective operand nodes.

A pseudo-code implementation of *FindPartialMatches* is given below.

```
FindPartialMatches()
{
  foreach subject operation node Nsub of Gsub
    foreach pattern Gpat in pattern library Patlib
      foreach pattern operation node Npat of Gpat
        if Nodematch(Nsub, Npat)
          // create a new match
          new Match(Nsub, Npat);
}
```

**Nodematch** The function *Nodematch* compares a subject operation node  $N_{sub}$  and a pattern operation node  $N_{pat}$  and returns whether *all* of the following criteria have been met:

- The opcodes of the operations are the same.
- If any input pattern operand  $O_{pat}$  is an immediate, then the corresponding subject operand  $O_{sub}$  should be an immediate with the same value.
- If any input pattern operand  $O_{pat}$  is connected to more than one input of  $N_{pat}$ , then the corresponding subject operand  $O_{sub}$  must be connected to the same set of inputs on  $N_{sub}$ .
- For each pair of corresponding results  $O_{sub}$  and  $O_{pat}$ :
  - The number of uses of  $O_{sub}$  is at least as high as the number of uses of  $O_{pat}$ .
  - If the number of uses of  $O_{sub}$  is greater than the number of uses of  $O_{pat}$ , then  $O_{pat}$  should be a pattern output, so that after substitution of the pattern into the trace (during covering), the value of  $O_{sub}$  is still available to the uses not covered by the pattern.
  - If  $O_{sub}$  is the current (valid) definition of its register (section 4.2), meaning that new uses could be added later on during the trace simulation, then  $O_{pat}$  must be an output of the pattern, so that after substituting the pattern into the trace, the value of  $O_{sub}$  is still available.

```

Nodematch(subject operation node Nsub,
           pattern operation node Npat)
{
    // check opcodes
    if(Nsub.opcode != Npat.opcode)
        return FALSE;

    // check input operand nodes
    foreach input operand Osub of Nsub/Opat of Npat
    {
        if(Opat.type == IMMEDIATE)
        {
            if(Osub.type != IMMEDIATE)
                return FALSE;
            if(Osub.imm_val != Opat.imm_val)
                return FALSE;
        }

        // check if shared pattern operands correspond
        // to shared subject operands
        if(Opat_checklist.Contains(Opat))
        {
            idx = Opat_checklist.GetIndex(Opat);
            if(Osub_checklist[idx] != Osub)
                return FALSE;
        }
        else
        {
            Opat_checklist.Append(Opat);
            Osub_checklist.Append(Osub);
        }
    }

    // check result operand nodes
    foreach result operand Osub of Nsub/Opat of Npat
    {
        if(Osub.num_uses < Opat.num_uses)
            return FALSE;
        if((Osub.num_uses > Opat.num_uses || Osub.is_current_def)
            && !Opat.IsPatternOutput())
            return FALSE;
    }

    return TRUE;
}

```

The abovementioned criteria are necessary for correctness. At some computational expense, additional criteria may be specified to reduce the number of partial matches that will be created because *Nodematch* is true, but which can never be merged into full matches. For example:

- If any of the inputs  $O_{pat}$  of  $N_{pat}$  has its defining operation node  $N_{pat-pre}$  within the pattern graph, then:
  - The corresponding  $O_{sub}$  should have a definer  $N_{sub-pre}$ .

–  $\text{Nodematch}(N_{\text{sub\_pre}}, N_{\text{pat\_pre}})$  is true.

**Match constructor** Whenever a match between a subject operation node  $N_{\text{sub}}$  and a pattern operation node  $N_{\text{pat}}$  has been found, a new match must be constructed. This is done by looking up the reference vector indices for all operand nodes  $O_{\text{pat}}$  connected to  $N_{\text{pat}}$ , then adding references to the corresponding subject operand nodes  $O_{\text{sub}}$  at the appropriate index. The reciprocal reference is set on  $O_{\text{sub}}$  by adding the  $(\text{Match}, \text{index})$  pair to its list of match references. A reference to the parent graph  $G_{\text{pat}}$  of  $N_{\text{pat}}$  is then added to the *Match*.

```
Match constructor(subject operation node Nsub,
                  pattern operation node Npat)
{
    // add input and output operand references
    foreach operand node Opat of Npat/Osub of Nsub
    {
        index = Opat.index;
        Match.ref[index] = Osub;
        Osub.AddMatchRef(Match, index);
    }

    Match.patterngraph = Npat.parentgraph;
}
```

## 5.2 Merging partial matches

After the partial match finding phase, we must now try to construct complete matches out of the partial ones that have just been generated. Looking at all the operand nodes  $O_{\text{sub}}$  of the subject graph  $G_{\text{sub}}$  (in no particular order), we look at all the matches found on  $O_{\text{sub}}$  and see which of these matches can be merged into new, higher degree partial or even complete matches.

**MergeMatches** The function *MergeMatches* looks at the matches found on each subject operand node  $O_{\text{sub}}$  and attempts to merge them with each other into new matches by calling the function *Merge* on them.

Note that the outer match loop (foreach  $M_1$ ) only considers the matches already on  $O_{\text{sub}}$  before any new, higher degree matches were added. The inner loop (foreach  $M_2$ ), however, considers all matches, including new ones, from  $M_1$  onwards. This is done for efficiency reasons; there are several ways to arrive at each higher degree match, but only one is needed.

```
MergeMatches()
{
    foreach subject operand node Osub of Gsub
    {
        // iterate only over matches already present
        foreach match M1 on Osub
        {
            // iterate including new matches from previous iterations
            foreach match M2 on Osub from M1 onwards
                Merge(Osub, M1, M2);
        }
    }
}
```

**Merge** Two matches  $M_1$  and  $M_2$  may be merged on operand node  $O_{sub}$  if they meet the following criteria:

- They refer to the same pattern graph.
- They both have the reference to  $O_{sub}$  in the same position of their reference vector.
- For each position of their reference vectors, if both  $M_1$  and  $M_2$  have a reference to a subject operand node in that position, those must refer to the same operand node. Otherwise the matches are said to *clash* at that position.

While the abovementioned criteria must be met for correctness, the following ensure that no redundant matches (matches equal to one already on  $O_{sub}$ ) are created:

- $M_1$  is not a subset of  $M_2$ , that is, merging  $M_1$  and  $M_2$  would not yield match equivalent to  $M_2$ .
- $M_2$  is not a subset of  $M_1$ .
- There is no match already on  $O_{sub}$  that is equivalent to the merger of  $M_1$  and  $M_2$ <sup>2</sup>.

If all criteria are met, a new match  $M_3$  can be created and a reference is appended to the match list on  $O_{sub}$ . Note that implementation shown below is slightly different.

The function *Merge* combines the merging of two matches  $M_1$  and  $M_2$  with checking for subset and clash conditions, for efficiency reasons: we now have to iterate along the reference vectors of the matches only once instead of four times (if we were to check each condition separately).

The function iterates over the reference vectors of  $M_1$  and  $M_2$  and finds which subject operand nodes  $O_{sub1}$  and  $O_{sub2}$  are referenced. For every pair of operand references, one of four situations may arise:

- $M_1$  references an operand node but  $M_2$  references nothing. We now know that  $M_1$  cannot be a subset of  $M_2$ .
- $M_1$  and  $M_2$  both reference an operand node. If those are different operand nodes, there is a clash: the merge is illegal.
- $M_1$  references nothing but  $M_2$  references an operand node. We now know that  $M_2$  cannot be a subset of  $M_1$ .
- $M_1$  and  $M_2$  both reference nothing.

If there is no clash, we put the reference in the current position of the reference vector of the new match  $M_3$ . After all references of  $M_1$  and  $M_2$  have been processed this way, we can check for the subset conditions. If either is still true, we reject the new match.

A redundancy check with all matches already present on  $O_{sub}$  is performed afterwards by calling the function *Redundant* with the newly formed match. The *Commit* function appends references to the new match to the appropriate subject operand nodes.

<sup>2</sup>This criterion encapsulates the previous two but requires more computation time to evaluate

```

// merge the two matches, if possible. Return FAILED otherwise
match Merge(operand Osub, match M1, match M2)
{
    // Check if pattern graphs are the same
    if(M1.pattern != M2.pattern)
        return FAILED;

    // Check if position in the matches is the same
    index = position of Osub in M1;
    if(M2[index] != Osub)
        return FAILED;

    // initialization
    M1_is_subset_of_M2 = TRUE;
    M2_is_subset_of_M1 = TRUE;

    // iterate along operand reference vectors
    foreach index in M1 // index for vector
    {
        Osub1 = M1[index];
        Osub2 = M2[index];
        if(Osub1 != NULL)
        {
            if(Osub2 == NULL)
                M1_is_subset_of_M2 = FALSE;
            else
                if(Osub2 != Osub1) // clashing operands
                    return FAILED;
            M3[index] = Osub1;
        }
        else // Osub1 == NULL
        {
            if(Osub2 != NULL)
                M2_is_subset_of_M1 = FALSE;
            M3[index] = Osub2;
        }
    }

    if(M1_is_subset_of_M2 || M2_is_subset_of_M1)
        return FAILED; // subset, new match would be redundant

    // Check for redundancy with all other matches on Osub
    if( Redundant(Osub, M3) )
        return FAILED;

    // Set references from operands to the new match
    Commit(M3);
    return SUCCEEDED;
}

```

**Redundant** checks all higher degree matches  $M_2$  on  $O_{sub}$  and looks for one equivalent to  $M_1$ . There are three conditions that have to be met for  $M_1$  to be equivalent to  $M_2$ :

- The patterns referenced by  $M_1$  and  $M_2$  are the same.
- The degree of  $M_1$  must be equal to the degree of  $M_2$ .
- $(M_1[index] == M_2[index])$  must hold for all values of  $index$ .

As soon as redundancy is detected, *Redundant* returns TRUE, since there is no need to look at further matches.

```
// Return TRUE if the operand node already
// contains a match just like this.
Redundant(Operand Osub, Match M1)
{
    foreach match M2 on Osub except M1
    {
        // must reference the same pattern
        if(M1.pattern != M2.pattern)
            continue;

        // degree must be the same
        if(M1.degree != M2.degree)
            continue;

        is_redundant = TRUE;
        foreach index in M1 // index for operand reference vector
        {
            Osub1 = M1[index];
            Osub2 = M2[index];
            if(Osub1 != Osub2)
            {
                is_redundant = FALSE;
                break;
            }
        }
        if(is_redundant)
            return TRUE;
    }
    return FALSE;
}
```

**Commit** The match  $M$  is not redundant with any of the matches already on  $O_{sub}$ , so we must add reciprocal references on the subject operand nodes it references.

```
// Add references to each of the operands
// the match reference vector contains.
Commit(Match M)
{
    foreach index in M // index for reference vector
    {
        Osub = M[index];
        if(Osub == NULL)
            continue;
        Osub->AddMatchRef(M, index);
    }
}
```

### 5.3 Cleaning up the partial matches

The final phase of the matching algorithm eliminates all incomplete matches, so that only full matches remain on all the subject graph nodes. This is done by comparing the degree of all matches to the number of operation nodes in the corresponding pattern graphs. If those are not equal, then the match is incomplete and must be deleted.

Another condition under which a match is deleted is that it would introduce a dependency cycle into the subject graph. This is the case if any of the matched pattern's outputs lead to any of its inputs, through some path in the subject graph.

```
// Remove all incomplete matches
CleanupMatches()
{
    foreach subject operand node Osub of subject graph Gsub
        foreach match M1 on Osub
            if(M1.degree != M1.patterngraph.num_operations
               || M1.CausesCycle())
                delete M1;
}
```

### 5.4 Integrated match creation and merging

In the previous subsections, the first and second phases of the matching algorithm, first degree match construction and match merging, have been described separately. However, it is possible to perform both tasks in a single iteration over the subject graph's operation nodes. This subsection describes the function *FindPartialAndMerge*, which is an implementation of integrated partial match finding and merging. In the top-level function *Matching*, it replaces *FindPartialMatches* and *MergeMatches*. Many of the function called in *FindPartialAndMerge* are described in the previous subsections.

#### FindPartialAndMerge

```
FindPartialAndMerge()
{
    foreach subject operation node Nsub
        foreach pattern graph Gpat
            foreach pattern operation node Npat
                FindPartialAndMerge(Nsub, Npat);
}

FindPartialAndMerge(subject operation node Nsub,
                    pattern operation node Npat)
{
    if(Nodematch(Nsub, Npat))
    {
        // first degree match construction
        M = new Match(Nsub, Npat);

        // match merging
        foreach input and output operand Osub of Nsub
        {
            // matches previously on this operand
            foreach match M1 on Osub up to M
```

```

    {
        // including newly added matches from
        // previous iterations of both M2 and M1
        foreach match M2 on Osub from M onwards
            Merge(Osub, M1, M2);
    }
}
}
}

```

## 5.5 Automatic library construction

The matching and covering algorithms described in the previous section operate using a predefined pattern library. In this section, we will look at an algorithm for the automatic detection and construction of patterns. Hooking into the matching algorithm's function *FindPartialAndMerge*, this algorithm operates by constructing new patterns out of previously detected (and matched) ones. In the following paragraphs, the function hierarchy of the automatic library construction algorithm will be described, starting with a modified version of *FindPartialAndMerge*.

**FindPartialAndMerge** This function is essentially the same as the one described in section 5.4, with one additional function call. Each time the partial matches on a subject operation node  $N_{sub}$  have been merged, the function *ConstructNewPatterns* is called, as shown below.

```

FindPartialAndMerge()
{
    foreach subject operation node Nsub
        foreach pattern graph Gpat
            foreach pattern operation node Npat
                FindPartialAndMerge(Nsub, Npat);
}

FindPartialAndMerge(subject operation node Nsub,
                    pattern operation node Npat)
{
    if(Nodematch(Nsub, Npat))
    {
        // first degree match construction
        M = new Match(Nsub, Npat);

        // match merging
        foreach input and output operand Osub of Nsub
        {
            // matches previously on this operand
            foreach match M1 on Osub up to M
            {
                // including newly added matches from
                // previous iterations of both M2 and M1
                foreach match M2 on Osub from M onwards
                    Merge(Osub, M1, M2);
            }
        }
    }
}

```

```

    // pattern construction
    ConstructNewPatterns(Nsub);
  }
}

```

**ConstructNewPatterns** This function considers all complete matches on the current subject operand  $O_{sub}$  and attempts to construct new matches by taking the pairwise union of those matches (*MatchUnion*). If this fails, we continue with the next pair of matches. If it succeeds, then the result of the union corresponds to a match with the new pattern we are trying to construct (an example of this is given in section 3.2).

We must now check if there already is a complete match with exactly the same operation and operand references (*RedundantMatch*). If this is the case, we know that the pattern corresponding to that match is equivalent to the one we are trying to construct. The new pattern would therefore be redundant, which means that we want to forget about constructing it and delete the match  $M_3$ .

If the aforementioned checks hold, then the new match  $M_3$  represents a new, unique pattern. We can now construct the corresponding pattern graph (*ConstructPattern*) and construct partial matches for  $M_3$ , for use in later merging operations (*ConstructPartialMatches*).

```

ConstructNewPatterns(subject operation node Nsub)
{
  foreach input Osub of Nsub
    foreach complete match M1 on Osub
    {
      foreach complete match M2 on Osub from M1 onwards
      { // excluding not matches added during this iteration of M1

        // try to take union
        M3 = MatchUnion(M1, M2);

        // check if MatchUnion succeeded
        if(M3 == NULL)
          continue;

        // check if resulting pattern already exists
        if(!RedundantMatch(Osub, M3))
        {
          // Build the new pattern and add to library
          ConstructPattern(M3);

          // Make partial matches from match M3
          ConstructPartialMatches(Nsub, M3.patterngraph);
        }
      }
    }
}

```

**MatchUnion** This function attempts to take the union of two matches  $M_1$  and  $M_2$  by taking the unions of their subject operation and operand reference vectors, and assigning them to the new match  $M_3$ .

Taking the union fails if  $M_1$  is a subset of  $M_2$  (or vice versa), in which case the resulting match would be redundant with  $M_1$  (or  $M_2$ , respectively).

The exception to this rule is when we try to take the union of a match with itself. This only happens if the match was referenced on the same operand node more than once, which in turn only happens if that operand node was referenced more than once by the match (i.e. used on more than one input of the match). This implies that a new pattern is possible, in which the multiplied operand is replaced by a single operand. This special case is handled separately, as can be seen in the pseudo-code implementation of *MatchUnion*, below.

For efficiency reasons, taking the union of the operation reference vectors of  $M_1$  and  $M_2$  and checking the subset conditions are integrated. First, all subject operation references of  $M_1$  are put into the subject operation reference vector of the new match  $M_3$ . If all the operation references that were added from  $M_1$  were also contained in  $M_2$ , then  $M_1$  is a subset of  $M_2$  and *MatchUnion* fails. After this, the subject operation references of  $M_2$  are added to  $M_3$ , if they weren't already contained in  $M_3$ . If no new references were added, then  $M_2$  is a subset of  $M_1$  and *MatchUnion* also fails.

If taking the union of the operation reference vectors succeeds, then the size constraints must be checked. If the number of operation references in the new match is greater than a predefined maximum number *MAX\_OPS\_IN\_PATTERN*, then the new pattern would become unacceptably large: *MatchUnion* fails.

If the size constraints are met, the union of the subject operand reference vectors of  $M_1$  and  $M_2$  can be computed. It is no longer necessary to check for subset conditions, since those would already have been detected when the operation reference vector union was computed. All references are 'uniquified': any operand node is referenced only once, even though it may have appeared more than once in the reference vectors of  $M_1$  or  $M_2$ . Note that, if desired, a size constraint on the number of operand references could be added after this.

```
match MatchUnion(match M1, match M2)
{
    // make empty match
    M3 = new match();

    if( M1 == M2 ) // merging with self?
    {
        M3.operationrefs = M1.operationrefs;
    }
    else
    {
        M1_subset_M2 = TRUE;
        M2_subset_M1 = TRUE;

        // add operation references from M1
        foreach Nsub in M1.operationrefs
        {
            M3.operationrefs.add(Nsub);
            if(!M2.operationrefs.contains(Nsub))
                M1_subset_M2 = FALSE;
        }

        if(M1_subset_M2 == TRUE)
            goto fail;

        // add operation references from M2
        foreach Nsub in M2.operationrefs
```

```

    {
        if(!M3.operationrefs.contains(Nsub))
        {
            M3.operationrefs.add(Nsub);
            M2_subset_M1 = FALSE;
        }
    }

    if(M2_subset_M1 == TRUE)
        goto fail:

    // check size constraints
    if(M3.operationrefs.count() > MAX_OPS_IN_PATTERN)
        goto fail:

    // add operand references from M1
    foreach Osub in M1.operandrefs
        if(!M3.operandrefs.contains(Osub))
            M3.operandrefs.add(Osub);
}

// add operand references from M2
foreach Osub in M2.operandrefs
    if(!M3.operandrefs.contains(Osub))
        M3.operandrefs.add(Osub);

return M3;

fail:
    delete M3;
    return NULL;
}

```

**RedundantMatch** The function *RedundantMatch* differs from the function *Redundant* in that the latter compares matches that are known to refer to the same pattern graph. *RedundantMatch* compares a match  $M$  for which there is no pattern graph yet with all complete matches on  $O_{sub}$  in order to determine whether an equivalent pattern graph already exists. If there is a match  $M_1$  on  $O_{sub}$  such that all of the following criteria are met, then the pattern graph corresponding to  $M$  is equivalent to the pattern graph corresponding to  $M_1$ :

1. The subject operation reference vectors of  $M$  and  $M_1$  are of equal length. This corresponds with the pattern graphs having an equal number of operations.
2. The subject operand reference vectors of  $M$  and  $M_1$  are of equal length. This corresponds with the pattern graphs having an equal number of operands.
3. All operations referenced in  $M$  are also referenced in  $M_1$ . This corresponds to the pattern graphs containing the same operations.
4. All operands referenced in  $M$  are also referenced in  $M_1$ . This corresponds to the pattern graphs containing the same operands.
5. All operands mapping to a pattern output in the pattern graph corresponding to  $M_1$  would also map to an output of the pattern graph corresponding to  $M$ .

The criteria are listed by order of the computational effort it requires to evaluate them. The further down the list, the more stringent the criteria become. Pattern graphs failing only the final criterion are isomorphic, except that they have different outputs: one or more values are used only internally in one of the patterns, while being made available externally (as outputs) in the other.

```

RedundantMatch(subject operand Osub, match M)
{
    // iterate over all complete matches
    foreach complete match M1 on Osub
    {
        // 1. Compare operation reference vector lengths
        if(M1.operationrefs.count() != M.operationrefs.count())
            goto next:

        // 2. Compare operand reference vector lengths
        if(M1.operandrefs.count() != M.operandrefs.count())
            goto next:

        // 3. Compare operation reference sets
        foreach Nsub in M1.operationrefs
            if(!M.operationrefs.contains(Nsub))
                goto next:

        // 4. Compare operand reference sets
        foreach Osub in M1.operandrefs
            if(!M.operandrefs.contains(Osub))
                goto next:

        // 5. Check pattern outputs
        foreach Osub in M.operandrefs
        {
            // determine if this would be a pattern output
            is_output = FALSE;
            foreach Nsub in Osub.uses
                if(!M.operationrefs.contains(Nsub))
                {
                    is_output = TRUE;
                    break;
                }

            // find corresponding pattern operand via M1
            Opat = M1.patterngraph.operands[Osub.matchref(M1).index];
            // if Osub is a pattern output in M, but not in M1
            // or vice versa
            if(Opat.IsOutput() != is_output)
                goto next:
        }

        // M1 and M are equivalent
        return TRUE;
    }

next:
}

```

```

    // all matches checked, no equivalent found
    return FALSE;
}

```

**ConstructPattern** Constructing the pattern graph corresponding to the match  $M$  is fairly straightforward. First, pattern operand nodes are created for each referenced operand node, and added to the pattern graph. If the subject operation defining the current subject operand is not referenced in  $M$  (i.e. lies outside the pattern), then the operand maps to a pattern input, which is marked as such. If there is any subject operation node that uses the operand but is not referenced in  $M$ , then the operand maps to a pattern output.

For each subject operation node referenced in the match, a pattern operation node is created and added to the graph. The operation's inputs and outputs are hooked up to it by adding def- and use-edges to the operands. The new graph is then added to the pattern library *PatLib*.

```

ConstructPattern(match M)
{
    Gpat = new pattern graph;

    // add pattern operand nodes
    for i = 1 to M.operandrefs.count()
    {
        Osub = M.operandrefs[i];
        Opat = new pattern operand("O{i}");
        Gpat.AddOperand(Opat);

        // determine if Opat is a pattern input
        Nsub = Osub.Def();
        if(Nsub != NULL)
            if(!M.operationrefs.contains(Nsub))
                Gpat.AddInput(Opat);

        // determine if Opat is a pattern output
        is_output = FALSE;
        foreach Nsub in Osub.uses
            if(!M.operationrefs.contains(Nsub))
            {
                is_output = TRUE;
                break;
            }
        if(is_output == TRUE)
            Gpat.AddOutput(Opat);
    }

    // add operation nodes
    foreach Nsub in M.operationrefs
    {
        Npat = new pattern operation(Nsub.opcode);
        Gpat.AddOperation(Npat);

        // hook up inputs...
        for i = 1 to #input operands of Nsub
        {

```

```

    Osub = Nsub.inputs[i];
    Opat = Gpat.operands[Osub.matchref(M).index];
    Npat.operand[i] = Opat;
    Opat.AddUse(Npat);
}

// ...and output
Osub = Nsub.output;
Opat = Gpat.operands[Osub.matchref(M).index];
Opat.AddDef(Npat);
}

// add to library
Patlib.AddPattern(Gpat);
}

```

**ConstructPartialMatches** This function constructs partial matches for the newly created pattern graph on all nodes of the subject graph that have already been processed (i.e. preceding  $N_{sub\_end}$ ). This is necessary to keep the partial matches consistent throughout the graph. The partial matches thus created may be merged into complete matches later (when processing nodes succeeding  $N_{sub\_end}$ ). If we neglected to create the partial matches, then instead of detecting a match with pattern  $G_{pat}$ , we would find a new pattern (equivalent to  $G_{pat}$  of course, but that fact would remain undetected).

```

ConstructPartialMatches(subject operation Nsub_end,
                        pattern Gpat)
{
    foreach operation Nsub in Gsub up to Nsub_end
        foreach operation Npat of Gpat
        {
            FindPartialAndMerge(Nsub, Npat);
        }
}

```

**IsCompletable** If the defining operation for a subject operand node is not present in the partial match, then that match can never become a complete match anymore. The operand node can never get a new definer in the subject graph, hence no new subject operation reference can be put into the partial match.

The function iterates over all pattern operation nodes of  $G_{pat}$ , the pattern graph corresponding to  $M$ . If there is a corresponding subject operation referenced in  $M$ , we want to check its input operands. If those correspond to pattern inputs, we cannot check anything. If they should have a defining operation referenced in  $M$ , but there is none, then  $M$  can never be completed and *IsCompletable* returns FALSE.

```

IsCompletable(match M)
{
    Gpat = M.patterngraph;
    foreach Npat of Gpat
    {
        // skip empty reference entries
        if(M.operationrefs[Npat.index] == NULL)

```

```

    continue;

    // check inputs for definers
    foreach input Opat of Npat
    {
        Osub = M.operandrefs[Opat.index];
        assert(Osub != NULL); // sanity check

        // no definer in graph? (== graph input)
        if(Gpat.inputs.contains(Opat))
            continue;

        // check if Osub's definer is referenced in M
        Npat_pre = Opat.Def();
        Nsub_pre = M.operationrefs[Npat_pre.index];
        if(Nsub_pre == NULL)
            return FALSE;
    }
    return TRUE;
}

```

## 6 Covering

After all possible matches between pattern graphs and the subject graph have been found by the matching algorithm (as described in section 5), we can now try to find a selection of matches to implement the subject graph with. This selection is called a *cover* of the subject graph, it must be chosen so that each node of the subject graph is covered by exactly one match. There are several possible covers, we are interested in the ones that minimize the latency of the longest path in subject graph.

The algorithm that is used for covering (finding a cover) is based on dynamic programming. It consists of the following phases:

1. **Evaluating the matches:** for each operand node in the subject graph, determine the data-ready time for all output matches that apply to it. Sort those matches by data-ready time, ascending.
2. **Sorting the graph outputs:** sort all subject graph output operands (operands that have no uses in the subject graph) by order of data-ready time of their best match, descending.
3. **Choosing a cover:** Starting at the subject graph output operand with the highest data-ready time for its best match, implement the match with the lowest possible data-ready time, recursively (depth-first). Mark the obtained data-ready times on all operand nodes. Terminate at subject graph inputs or already covered operand nodes.

```

Cover()
{
    EvaluateMatches();
    SortGraphOutputs();
    CoverOutputs();
}

```

The first phase, evaluating the matches, is described in section 6.1. The second phase is just a sorting operation and will not be described in detail. The third phase, choosing a cover from the available matches, is described in section 6.2.

## 6.1 Evaluating the matches

The matches on each subject graph operand have to be evaluated in terms of the data-ready time they would give the operand, when implemented. This is done by a hierarchy of functions, of which *EvaluateMatches* is the top one.

**EvaluateMatches** This function iterates over all subject graph operands, in no particular order, and evaluates the matches on each subject operand node. The reason that no particular order is required is that the recursive function *EvaluateMatches* stores its results on the subject operand node *Osub*. Therefore, if during any iteration the same node is visited more than once, the function simply returns the results that were calculated on the first visit. This eliminates the need for a topological sort.

```
EvaluateMatches()
{
    foreach subject operand node Osub
        TimeOutputMatches(Osub);
}
```

**TimeOutputMatches** The purpose of this function is to determine the data-ready time for operand *Osub* that each of its matches would give it, when implemented. The matches are sorted by this quantity.

First we check if the subject graph operand node *Osub* is an input of the subject graph, that is, if it has no defining operations in the graph. If this is the case, we return the node's data-ready time, as calculated during a previous covering iteration (or zero if the current covering iteration is the first).

Then we check if this node *Osub* has been visited in a previous iteration. This is done by checking for entries in the output match reference list. If there are none, we obviously haven't called *FindOutputMatches* yet, so we must execute the rest of the function. If there are, we simply return the previously calculated best data-ready time for output matches on *Osub*.

We then determine the data-ready time for each of the output matches. This is done by taking the maximum of the latency of the pattern from any of the pattern inputs to the current operand, added to the data-ready time of the input. After all output matches have been timed, they are sorted according to data-ready time, ascending.

```
int TimeOutputMatches(subject operand node Osub)
{
    // Check if Osub is a subject-graph input
    if(Osub.IsSubjectGraphInput())
        return Osub.data_ready_time;

    // Check if we have already visited this Osub.
    Mref = first output match reference of Osub;
    if(Mref != NULL)
        return Mref.data_ready_time;

    FindOutputMatches(Osub);
}
```

```

max_data_ready = 0;

// find data-ready time for all output matches
foreach output match reference Mref on Osub
{
    match M = Mref.match;
    index = Mref.index;
    Gpat = M.patterngraph;
    Opat_out = Gpat.operands[index];

    // time for each match input
    foreach pattern input operand Opat_in of M
    {
        pat_latency = Gpat.latency(Opat_in, Opat_out);
        input_index = Opat_in.index;
        Osub_in = M[input_index];
        data_ready = TimeOutputMatches(Osub_in) + pat_latency;
        max_data_ready = Max(max_data_ready, data_ready);
    }

    Mref.data_ready_time = max_data_ready;
}

// sort output match references according to data-ready time
foreach output match reference Mref1 on Osub
    foreach output match reference Mref2
        on Osub from Mref1 onwards
        if(Mref1.data_ready_time > Mref2.data_ready_time)
            Swap(Mref1, Mref2);
}

```

**FindOutputMatches** determines which matches on the operand are so-called output matches (see also section 3.3). This is fairly straightforward: the reference to the match can be used to find the pattern operand node that the current subject operand node corresponds to. If this pattern operand is a pattern output, then the match is an output match to the current subject operand.

```

FindOutputMatches(subject operand node Osub)
{
    foreach match reference Mref on Osub
    {
        match M = Mref.match;
        index = Mref.index;
        Gpat = M.patterngraph;
        Opat = Gpat.operands[index];
        if(Opat.IsOutputNode)
            Osub.AddOutputMatchRef(Mref)
    }
}

```

## 6.2 Choosing a cover

After all output matches on all subject graph operand nodes have been evaluated and sorted in terms of best attainable data-ready time, the time has now come to make a selection from those output matches: the cover. This is done by calling the top-level function *CoverOutputs*.

**CoverOutputs** This function iterates over the subject graph output operands, which have been sorted by projected data-ready time, descending, in the previous phase. It calls the recursive function *ImplementBestMatch* for each of them.

```
CoverOutputs()
{
    foreach subject graph output operand Osub (sorted)
        ImplementBestMatch(Osub);
}
```

**ImplementBestMatch** First we check if the current node has already been covered as part of a earlier covering iteration. If this is the case, we return the attained data-ready time from that cover.

The best available matches on the subject sub-graph rooted in node  $O_{sub}$  are implemented, recursively. The best available match is the first one on  $O_{sub}$  that would not introduce any dependency cycles into the subject graph.

To guarantee that the critical path to the current subject operand node is covered first, we have to determine the precedence with which the subject operand nodes corresponding to the match inputs are covered. This is done by calculating the *slack* for each one of these nodes. The slack is defined as the difference between the time a match input should be ready (the current match output's data-ready time minus the pattern latency) and the data-ready time of the input's best match. The match inputs are sorted by slack, ascending, before covering them in that order.

The attained data-ready time is calculated and marked on the subject operand nodes. All nodes covered by the selected matches are marked as having been covered by the function *SelectMatch*.

```
int ImplementBestMatch(subject operand node Osub)
{
    index = Mref.index;
    Gpat = M.patterngraph;
    Opat_out = Gpat.operands[index];

    // check if we've already covered this node
    if(Osub.has_been_covered == TRUE)
        return Osub.data_ready_time;

    // find the best match that can be implemented
    foreach output match reference Mref of Osub
        if(Mref.match.IsImplementable())
            break;
    M = Mref.match;

    SelectMatch(M);

    // determine precedence
    output_time = TimeOutputMatches(Osub);
```

```

foreach pattern input operand Opat_in of M
{
    Osub_in = subject node corresponding to Opat_in;
    pat_latency = Gpat.latency(Opat_in, Opat_out);
    input_time = TimeOutputMatches(Osub_in);
    Osub_in.slack = output_time - pat_latency - input_time;
}

sort pattern input operands Osub_in by slack, ascending;

// calculate ready-time
foreach pattern input operand Osub_in, sorted
{
    Opat_in = pattern node corresponding to Osub_in;
    pat_latency = Gpat.latency(Opat_in, Opat_out);
    data_ready = ImplementBestMatch(Osub_in) + pat_latency;
    max_data_ready = Max(max_data_ready, data_ready);
}

Osub.data_ready_time = max_data_ready;
return max_data_ready;
}

```

**SelectMatch** The match selected for implementation probably overlaps several other matches. These can never be implemented anymore, so they must be deleted. This function removes all other matches that are referenced on the operand nodes in the selected match.

Note that any of the removed matches may have been the best match for another operand node. This means that the best implementation, the one that match evaluations for other operands are based on, is no longer available there. The impact of removing any match may resonate throughout the subject graph. Keeping track of these effects is not trivial and would require a lot of extra computational effort (each time a match is implemented, we have to re-evaluate all matches on all nodes). We therefore settle for staying with our old match evaluation results and implement the next best available match, as determined previously, everywhere.

```

SelectMatch(match M)
{
    foreach index in M // index for operand reference vector
    {
        Osub = M[index];
        Osub.has_been_covered = TRUE;

        foreach match M_not_taken on Osub
        {
            // delete matches that overlap with M
            if(M_not_taken != M)
                delete M;
        }
    }
}

```

## 7 Conclusions

In this report, new matching and covering strategies were presented. The matching algorithm can handle pattern graphs of arbitrary topology, as long as each pattern's operation and operand nodes are all connected in some way (i.e. SIMD-style patterns are not detected).

The covering algorithm is based on dynamic programming. When there are only tree-shaped patterns in the pattern library and the subject graph is a tree as well, it behaves exactly like the dynamic programming algorithm and therefore produces optimal results. If more complex patterns are used, the covering algorithm is heuristic and optimality cannot be guaranteed.

## References

- [1] K. Keutzer. Dagon: Technology binding and local optimization by dag matching. In *DAC, Proceedings of the Design Automation Conference*, pages 617-623, May 1987.
- [2] Yuji Kukimoto, Robert K. Brayton, and Prashant Sawkar. Delay-optimal technology mapping by dag covering. In *Proceedings of the Design Automation Conference*, 1998.
- [3] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGRAW-HILL, 1994.
- [4] R. Rudell. Logic synthesis for vlsi design. Technical Report UCB/ERL M89/49, University of California at Berkeley, April 1989.

# Instruction Set Synthesis Using Operation Pattern Detection

Marnix Arnold and Henk Corporaal

Computer Architecture Laboratory  
Department of Electrical Engineering  
Delft University of Technology  
{marnix, heco}@cardit.et.tudelft.nl

**Keywords:** pattern matching, co-design, design space exploration, instruction set synthesis.

## Abstract

*An important problem in the area of processor design for embedded systems is determining the proper instruction set architecture. Trade-offs have to be made between programmability and reusability of dedicated hardware for special functionality on the one hand, and a high performance dedicated instruction set on the other hand. This paper addresses the question of how to find specialized instruction set extensions for a set of applications.*

*We describe the application of a new pattern matching technique to the problem of the identification of recurring patterns of operations. By implementing frequently occurring operation patterns in hardware, and using this hardware as special function units, a fine-grained hardware/software partitioning can be found. The fine granularity, and the fact that patterns are taken from a number of different target applications rather than a single one, increase the opportunities for reuse of the special-purpose hardware. We illustrate our technique with experiments on a number of benchmarks from the DSP domain.*

## 1 Introduction

Hardware/software co-design is often performed on a per-application basis, yielding systems that are highly application-specific. The approach taken is usually a coarse-grained one: entire functions of the application are mapped either in hardware or software [4]. Any hardware thus generated is only reusable by other applications if those include the same function. For

application-specific systems this is obviously not a problem. For application-domain-specific, re-programmable systems, however, we may want to increase the granularity of the hardware/software partition, to increase the chances of hardware reuse. It is with this in mind that we consider a partitioning approach that centers on groups of instructions rather than entire functions. In this paper, we present a new algorithm for the automatic, on-the-fly detection of groups (patterns) of instructions as they occur in the application(s) that we want to generate an execution engine for. Patterns that occur frequently among the target applications can then be considered for implementation in hardware.

Section 2 discusses work from the related areas of instruction set synthesis, technology mapping and code generation. An overview of the algorithm used for the detection of recurring operation patterns is given in section 3. Experiments and their results are described in section 4. We conclude this paper with section 5.

## 2 Related Work

Pattern matching techniques have been around for quite some time, originating from the string matching problem [1]. Keutzer [5] first applied pattern matching techniques to the problem of technology mapping, noting similarities with the code generation problem [2]. Recent work by Kukimoto [6] extended these techniques to allow for rooted-DAG-shaped subject graphs, as opposed to tree-shaped graphs. However, no matching techniques are available that deal with patterns that are not trees or single-output DAGs [8]. We would like to detect and exploit such patterns, though, so we were forced to come up with a new

matching algorithm [3].

The work in this paper is somewhat related to the field of instruction set synthesis. Liem et al. [7] use matching and covering techniques to identify recurring instances of patterns from a predefined library, rather than constructing this library automatically on-the-fly, as we will do. The search for new operation patterns as described in [9] is limited to chains (sequences) of operations, whereas we consider patterns of any shape.

### 3 Algorithm Overview

For the matching of pattern graphs that are not trees, conventional techniques are not suitable. For this reason, we have come up with a new matching algorithm, based on the principle of *incremental matching*. In this section we will give an overview of the matching algorithm and how it can be extended to automatically construct a library of recurring patterns. The covering algorithm we employ in section 4 will only be discussed briefly. A comprehensive description of the algorithms can be found in [3].

#### 3.1 Incremental Matching

Given a subject graph  $G_{sub}$  and a library of pattern graphs  $PatLib = \{G_{pat_i}\}$ , as shown in figure 2, we iterate over all operation nodes in  $G_{sub}$  (nodes I, II and III), in no particular order. Each of the subject graph's operation nodes  $N_{sub}$  is matched against all the pattern graphs' operation nodes  $N_{pat}$ . A pair of operation nodes ( $N_{sub}, N_{pat}$ ) match if they meet all of the following criteria:

- The opcodes of  $N_{sub}$  and  $N_{pat}$  are the same.
- For the pair of result operands ( $O_{sub}, O_{pat}$ ) of  $N_{sub}$  and  $N_{pat}$ , respectively:
  - $NumUses(O_{sub}) \geq NumUses(O_{pat})$ .
  - $NumUses(O_{sub}) = NumUses(O_{pat})$ , unless  $O_{pat}$  is a pattern output.
- For each pair of corresponding inputs ( $O_{sub}, O_{pat}$ ) of  $N_{sub}$  and  $N_{pat}$ , respectively<sup>1</sup>:
  - $NumUses(O_{sub}) \geq NumUses(O_{pat})$ .
  - $NumUses(O_{sub}) = NumUses(O_{pat})$ , unless  $O_{pat}$  is a pattern input or output (not an internal operand node).

<sup>1</sup>For commutative operations, more than one pair can be formed for each input. These pairs are checked separately.

- For each pair of corresponding inputs ( $O_{sub}, O_{pat}$ ) of  $N_{sub}$  and  $N_{pat}$ , respectively, if  $O_{pat}$  is an immediate value (literal) rather than a register reference, then  $O_{sub}$  must be an immediate of the same value. Note that the converse does not have to hold, since it is always possible to map the value of an immediate ( $O_{sub}$ ) into a register ( $O_{pat}$ ).

where  $NumUses(a)$  denotes the number of times operand node  $a$  is used as an input to an operation node.

Figure 1 illustrates how the matching criteria can keep the number of partial matches under control (we will show later why this is important). If we consider the pair of operation nodes ( $I, n_1$ ) in figure 1a, we see that  $NumUses(c) < NumUses(o_4)$ , which violates the second criterion. Also, if we consider the pair ( $II, n_3$ ), we see that for the operand pair ( $c, o_4$ ) the fourth criterion is violated. Violations of the third and fifth criterion are illustrated in figure 1b, where  $NumUses(d) > NumUses(o_3)$ , but  $o_3$  is an internal operand node.

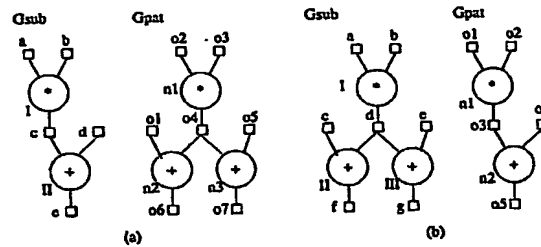


Figure 1: Examples of graphs failing the matching criteria.

In figure 3, the *partial matches*  $m_1$  and  $m_2$  constructed for pattern graph  $G_{pat4}$  are shown (there would, of course, also be partial matches for the other patterns). As can be seen, a match is really nothing more than two reference vectors that refer to subject operations and operands, respectively. The position a reference occupies in a reference vector indicates with which pattern operation (operand) it corresponds to.

Looking at the partial matches shown in figure 3, we notice that the matches  $m_1$  and  $m_2$  share the reference to subject operand node  $c$ . These two matches can be combined into a new match  $m_3$ , which is the union of the two. This match has no empty spaces in its reference vectors, and is said to be *complete*, otherwise it is still a partial match. In the same manner, we can combine partial matches for other patterns into complete matches, regardless of the size and shape

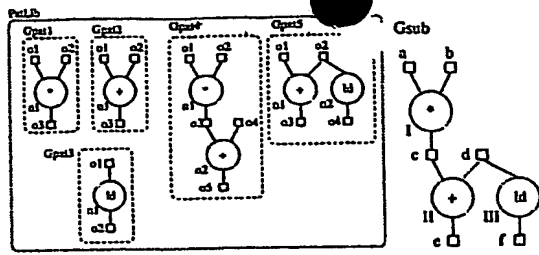


Figure 2: Example of pattern and subject graphs.

of those patterns. Indeed, pattern graph  $G_{pat5}$ , which is a multiple-output DAG, is matched in the exact same way as graph  $G_{pat4}$  from our example.

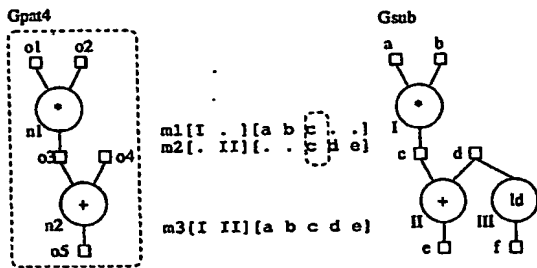


Figure 3: Partial matches  $m_1$  and  $m_2$  merge into the complete match  $m_3$ .

A note on the complexity of the matching algorithm: each of the subject graph's operation nodes is matched against each of the pattern graphs' operation nodes. New, partial matches are created each time an  $(N_{sub}, N_{pat})$  pair matches. Subsequently, the newly created partial match is checked against *all* available partial matches (referring to the same pattern) on the current  $N_{sub}$ 's input (one or two) and output (zero or one) operands. Whenever the new match can be successfully merged with another match, a new, more complete match is created (c.f. match  $m_3$  in figure 3). We then have to check if we did not already construct this match in some other way (by merging a different pair of partial matches), by checking its equivalence against, again, all matches on the current  $N_{sub}$ 's inputs and outputs. If it is not equivalent to any existing match, we can add the new match to the graph.

The complexity of the matching algorithm can be formulated as:

$$\mathcal{O}(N_s \times N_p \times (1 + P_{mt} \times M \times (1 + P_{mg} \times M))) \quad (1)$$

where: 6044539.020503

- $N_s$  is the number of operation nodes in the subject graph.
- $N_p$  is the number of operation nodes in all pattern graphs.
- $P_{mt}$  is the chance of a match between the current subject and pattern operation nodes.
- $P_{mg}$  is the chance of a successful merge of the newly created partial match and the current existing match.
- $M$  is the number of matches present on the operands of the current subject graph operation node.

In the worst case, the complexity of the algorithm is quadratically dependent on  $M$ . In order to minimize the computational effort that has to be spent in the inner loop, as well as the memory required to store all the partial matches, it is important that the number of partial matches that are created is kept to a minimum<sup>2</sup>. It is for this reason that we use the rather elaborate set of matching conditions stated earlier in this section.

### 3.2 Library Construction

In the previous section, we saw that our new matching algorithm can handle patterns of any size and shape, whereas conventional matching algorithms only operate on tree-shaped patterns. Another drawback of using a conventional matching algorithm is that it requires the pattern library to be defined beforehand. This implies that we need some advance knowledge of which patterns to expect, but such knowledge may not be available. We will show that this problem can be overcome by extending the incremental matching algorithm to include automatic library construction capabilities.

As before, we start with a subject graph  $G_{sub}$  and a library of pattern graphs  $PatLib = \{G_{pat_i}\}$ , as shown in figure 4. The difference with the previous example is that the initial pattern library now only contains a minimal set of patterns: just the ones with a single operation node. When we start processing the subject graph's operation nodes, we will initially only construct matches for the single-operation pattern graphs. Whenever we finish constructing partial matches on an operation node, we can now look for opportunities to create new pattern graphs.

<sup>2</sup>Note that  $P_{mg}$ , the chance of a merge of two matches, decreases if there is an increase in the number of 'unmergeable' matches due to a large number of pattern graphs.

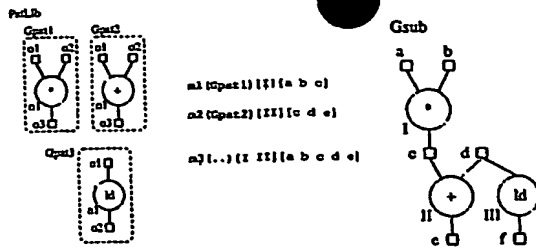


Figure 4: Pattern construction by combining matches.

In the example of figure 4, after we finish processing operation nodes *I* and *II* (again in an arbitrary order), we see that two matches  $m_1$  and  $m_2$  both contain a reference to operand node *c*. Note that now the matches refer to different patterns, which was not the case in the example of figure 3. By combining the matches  $m_1$  and  $m_2$  into a new match  $m_3$ , we have a recipe for constructing a new pattern! By copying all the nodes (operations and operands) referenced in  $m_3$  into a new pattern and adding it to the library, we will be able to log all future occurrences of this pattern.

It is possible that a pattern constructed in this way is already in the library. This can be checked by comparing the new match ( $m_3$ ) to all complete matches on the current operand node (*c*). If a match is found that references the exact same operand nodes as  $m_3$ , the pattern it refers to must be equivalent to the newly constructed one. Adding the new pattern to the library would only add redundancy and more matches, which we want to avoid; we therefore discard the new pattern and its match  $m_3$ .

In principle, it is possible to detect patterns of arbitrary size. However, the number of possible patterns in the subject graph explodes when we allow large patterns. In addition, the chances of a pattern recurring in a graph drop as its size goes up, so there is a limit to the usefulness of larger patterns. Still, the number of patterns that can be found if the pattern size is only two or three operation nodes is considerable. Furthermore, a large number of patterns causes a large number of partial matches, each of which have to be considered in the inner loop of the matching algorithm (section 3.1). It is easy for the matching algorithm to become bogged down in partial matches that may never become complete. For this reason, we decided to keep track of only a limited number of patterns. A pattern buffer of 200 patterns is kept, in which the patterns are sorted according

to how often they occur in the subject graph up to that point. Each time the buffer becomes full, the least frequently occurring half of the patterns is discarded. That way, newly constructed patterns have some time to climb into the top 50% before the next purge. A drawback of this method is, that any recurrence of a discarded pattern is later seen as the occurrence of a new pattern.

### 3.3 Covering

When the matching algorithm finishes, all occurrences of library patterns in the subject graph have been detected and marked. We now want to find a *cover*, a subset of the total set of detected matches, such that:

- All operation nodes in the subject graph are included in exactly one match of the cover.
- A certain characteristic of the cover is optimized. This can be the number of matches in the cover, or the longest path's length, or a combination of both.

Note that it is not strictly necessary for each operation node to be included in exactly one match. It may be profitable to include overlapping matches in the cover, duplicating operation nodes, in order to reduce the length of the graph's critical path [6], at the expense of an increase in the number of matches in the resulting cover (meaning, in this case, chip area). However, since we aim at reducing the operation count of our graph (the number of matches in our cover), we thought it prudent not to allow node duplication.

The heuristic we use to select a cover operates on the premise that reducing the number of operations in all paths of the graph will also reduce the total operation count. In order to minimize the number of operations in a path, we apply an algorithm that is similar to dynamic programming [5], but differs in that it handles reconvergent paths in the graph.

Our algorithm starts by determining, for each subject graph operand node, the matches for which the operand maps to an output of the corresponding pattern graph. These matches are called the *output matches* of that operand. The output matches are then sorted ascendingly, on each operand node, according to the number of matches in the longest path leading up to it. After all operand nodes have been processed this way, we can determine which of the subject graph's output operands has the longest path leading up to it. We implement (select for the cover) the best match on that operand, and invalidate all

matches that overlap with it (we refer to operation or operand nodes that are also referenced in this match). We then recursively do the same for all paths leading up to the selected match, by order of path length. Reconvergence in the paths is handled by terminating the recursion whenever we come across an operand node which has been covered already (for which a match has been selected already). After all paths leading up to the current subject graph output operand have been covered this way, we can handle the other subject graphs outputs in the same fashion.

## 4 Experiments

We execute our method for detecting recurring patterns of operations on a number of well-known benchmarks from the DSP domain. In this paper, we limit the size of the patterns to two operation nodes, even though the pattern detection and matching algorithms can handle patterns of arbitrary size. An overview of the benchmarks, with their dynamic operation counts, is given in table 1.

Name	Description	#ops
bspline	FIR Filter	6149
compress	Compression (dct 2d)	163513
dft	Discrete FFT	6666
edge	Edge detection	268717
expand	Decompression (idct 2d)	151083
foewf	5th Order Elliptic Wave	13067
fir	35 pt. Lowpass FIR	30459
flatten	Level histogram of image	33960
iir	IIR highpass filter	10794
pse	Sehwa's FIR filter	6917
smooth	Convolution w. 3x3 kernel	83365

Table 1: DSP benchmarks.

Each benchmark is trace simulated, and the pattern detection experiments are performed on the dataflow graph of the execution trace. The detected patterns for all benchmarks are then put into a unified pattern library. This library is then used to cover the execution trace of each benchmark, to get a feeling for how much each pattern would help reduce the operation count of that benchmark. These coverings can be seen as the best results the covering algorithm can attain with an unbounded pattern library (after all, all patterns ever detected in any of the benchmarks are there).

The reason we use an execution trace rather than the static object code for our experiments is that a trace effectively masks control flow, making pattern matches visible that reach across ba-

sic block or even to boundaries. If we overlay the trace with the patterns we found, then the matches that cross control flow boundaries would seem to indicate that code motion (speculative execution, loop unrolling, etc.) could be beneficial.

**Constructing a unified library** After covering, we calculate how often each pattern was used for each benchmark. Note that it is misleading to just count the matches for that pattern, as it is unknown how many of those (likely to be mutually exclusive) matches will be chosen for the cover. The patterns can now be sorted according to how much they contributed to the unbounded-library covering (i.e., as a percentage of the total number of matches that were chosen for the cover). Since we are most interested in patterns that contribute to the operation count reduction of the entire application domain, rather than just one application, we sort the patterns by the average of their contributions to the per-benchmark coverings. The reason we average the contributions of the patterns (a percentage) rather than the absolute share in the coverings (a number of pattern instantiations) is that we do not want to skew our results towards the larger benchmarks. This yields a unified, sorted pattern library in which all benchmarks are represented equally, independent of the size of their respective data sets.

**Unbounded library covering analysis** In figure 5, we see the (cumulative) contribution to the individual benchmarks' coverings of pattern libraries that consist of the top- $x$  patterns of the unified library. As can be seen, some benchmarks get a better-than-average contribution (bspline, pse) and some get a worse-than-average contribution (foewf). This can be interpreted as follows: the more an application's contribution graph pulls towards the upper-left corner of the figure, the more 'average' or representative for the application domain the application is. As a consequence of this, figure 5 can also be used to judge how well the applications fit together on the same instruction set, something which is difficult to determine by inspection of the benchmarks' source code alone. A final remark on figure 5: it can be seen that from the top-80 patterns onward, there is no additional contribution to the covering of any of the benchmarks (the 100% mark has been reached). This implies that none of the patterns added to the library from that point onward are ever actually used in the coverings of any of the benchmarks.

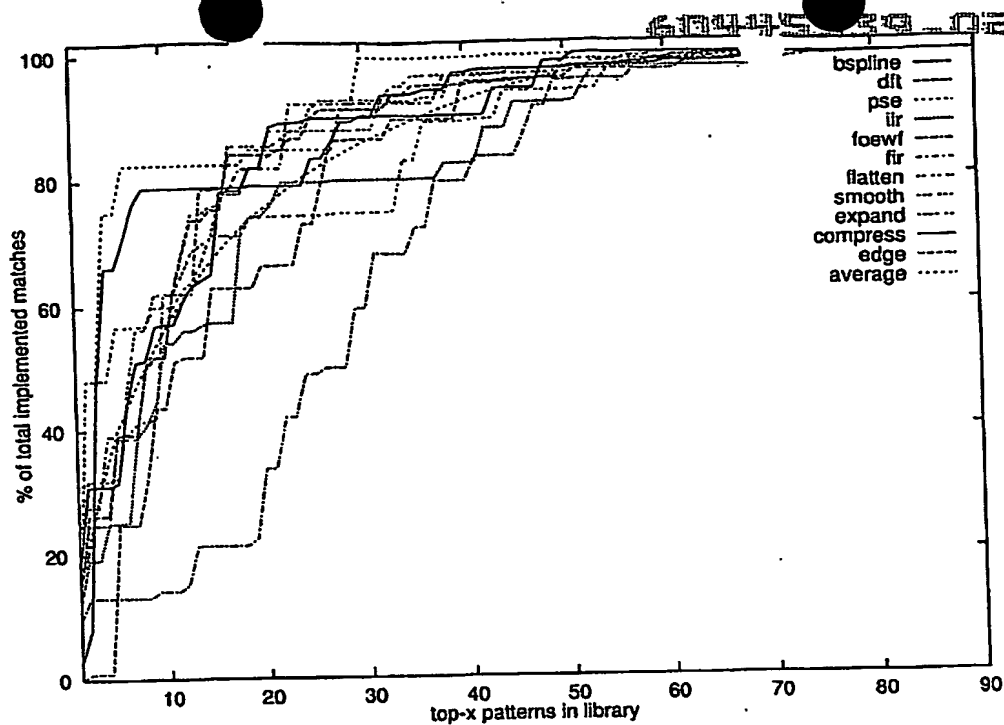


Figure 5: The contribution of patterns to the coverings.

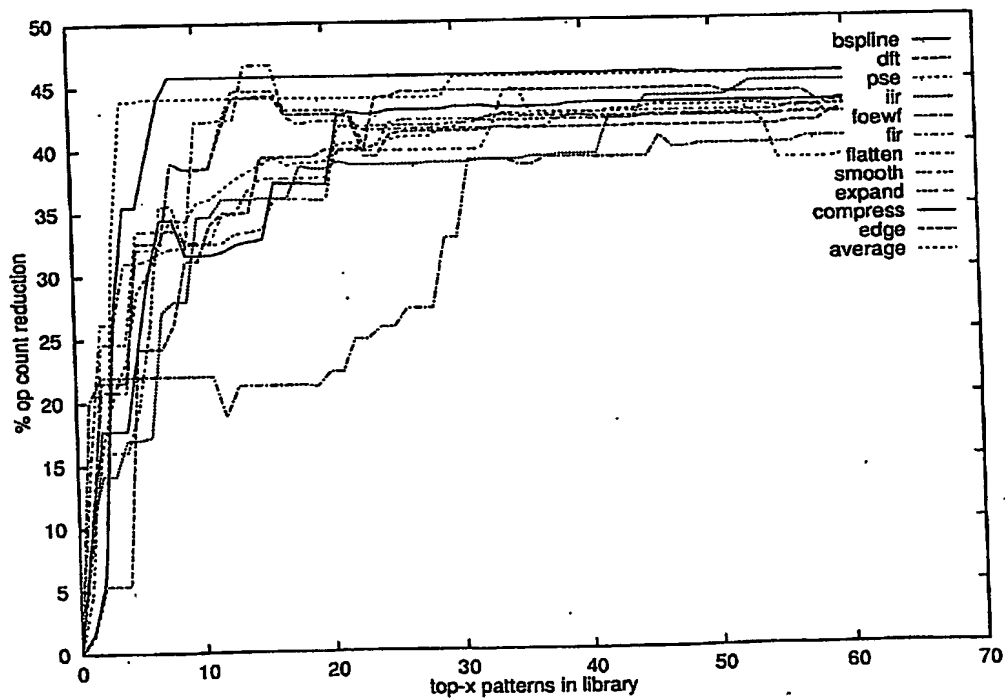


Figure 6: Operation count reduction for the incremental pattern libraries.

**Covering with partial library.** Now that we have found a ranking of patterns in the unified library, we can check if there is a relation between the contribution of the patterns in the top-x libraries to the 'ideal' (unbounded library) covering, and the actual results of the covering algorithm for each of the top-x libraries. For this, it is necessary to re-cover each of the benchmarks using each of the top-x pattern libraries. The results of these coverings can be found in figure 6. It can be expected that if a library has a lower-than-average contribution (in figure 5), then the operation count reduction will also be below average. This is confirmed by the curve for the foewf benchmark. Similarly, benchmarks with a higher-than-average contribution should have a higher-than-average operation count reduction. The operation count reduction curves for the bspline and pse benchmarks confirm this.

Note that the curves in figure 6 do not increase monotonously, which is what we would expect if we give the covering algorithm an extra pattern to work with each time. These occasional dips are due to the fact that the covering algorithm is a heuristic method, which every once in a while gets confused and yields a sub-optimal result. Figure 7 illustrates how the addition of a pattern  $G_{pat3}$  actually worsens the cover: initially, as only patterns  $G_{pat1}$  and  $G_{pat2}$  are available, the covering algorithm will apply  $G_{pat2}$  twice, on operations  $\{I, II\}$  and operations  $\{III, IV\}$ , arriving at a cover of size two. After pattern  $G_{pat3}$  becomes available, however, the covering algorithm will assume that reducing the number of matches in all paths will also reduce the total size of the cover. It therefore applies  $G_{pat3}$  to operations  $\{II, IV\}$ , shortening the path through operations  $II$  and  $IV$  to one match. This leaves operations  $I$  and  $III$  to be covered by  $G_{pat1}$ , resulting in a cover of three matches.

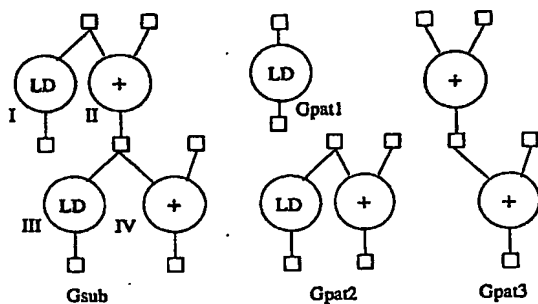


Figure 7: Adding patterns may worsen the cover.

Also note that 50% operation count reduc-

tion is a hard limit. For the purposes of this experiment, we only use patterns that are no larger than two operations, the best result we can theoretically get is obtained if all operation are replaced, in pairs, with two-operation patterns. In addition, it must be noted here that, since the covering algorithm operates on execution traces and hence ignores control flow, the operation count reduction figures must be seen as upper bounds for the operation count reduction that a compiler can achieve when performing code generation (when control flow is taken into account).

**Analysis of the top 10 patterns** The top ten patterns of the unified library are shown in figure 8. The most popular pattern (nr.1) is an integer add, followed by another integer add. In hardware, this can be implemented as an add, followed by an accumulate on the same unit, or as a 3-input, 2-output unit that can execute the pattern in a single processor cycle [10]. The second pattern is a conditional jump, where the condition is calculated by the greater-than node. Patterns 5 and 7 are array references, where the address calculation consists of a base-plus-offset calculation. Pattern 6 is the well-known multiply-add, pattern 8 the almost equally well-known add-shift. Note that pattern 9 performs the same calculation on both nodes! It looks as if the compiler missed an optimization opportunity, possibly hidden by control flow.

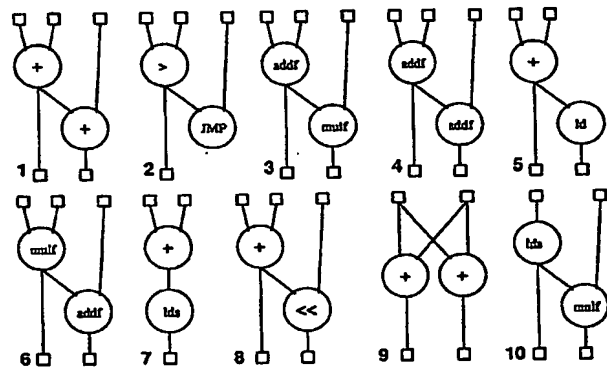


Figure 8: The top 10 patterns.

It can be seen that the top 10 patterns are not proper trees, since most also export their intermediate values. However, with the exception of pattern 9, none of the patterns represent operations that execute in parallel. It is quite possible that this is an artifact of the covering heuristic, which favors chains of operations. It will be in-

interesting to see whether this operation changes if we come up with a different covering strategy.

## 5 Conclusion and Future Work

We presented a technique for identifying common operation patterns across a range of applications, using a new pattern matching algorithm. This algorithm is innovative in that it can handle patterns of arbitrary shape, widening the scope of the search for operation patterns that can be implemented in hardware. Newly discovered patterns are added to the pattern library on-the-fly, resulting in a single pattern detection (finding new patterns) and matching (marking occurrences of patterns) pass.

Using the new technique, we found patterns of operations common to a set of benchmarks, which, when covering is applied, indicate that a substantial operation count reduction is possible (e.g., 20 patterns yield an average operation count reduction of 40%). Furthermore, we were able to incrementally construct a library of new operations (patterns) and analyze the influence of each new operation on the average operation count as well as on the operation count of each benchmark separately.

The covering heuristic, which was based on dynamic programming, still leaves something to be desired. The influence of the covering algorithm on the selection of the patterns of various shapes is not well understood at this point. Different algorithms may yield different top-x pattern libraries, which is something that needs to be investigated.

It has already been noted that our method operates on (dynamic) execution traces rather than (static) code. The absence of control flow in an execution trace can be seen as both an advantage and a disadvantage. Patterns invisible in static code can be detected dynamically, but exploiting these patterns during static code generation may prove difficult, requiring various code transformations, such as loop unrolling, speculative execution, etc. These transformations may cause an increase in code size, nullifying the effect of the patterns. In the future, we will concentrate on how to extend our current techniques to the problem of code generation.

## References

- [1] A. Aho and M. Corasick. Efficient string matching: An aid to bibliographic research. *Communications of the ACM*, 18(6):333-340, June 1975.
- [2] Alfred Aho, J. E. Hopcroft, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.
- [3] Marnix Arnold and Henk Corporaal. Matching and covering with multiple-output patterns. Technical Report 1-68340-44(1999)-01, Delft University of Technology, <http://cardit.et.tudelft.nl/MOVE/documents/Arnold99a.ps>, 1999.
- [4] Peter M. Athanas and Harvey S. Silverman. Processor reconfiguration through instruction set metamorphosis. *IEEE Computer*, (0018-9162/93/0300-0011):11-18, 1993.
- [5] K. Keutzer. Dagon: Technology binding and local optimization by dag matching. In *DAC, Proceedings of the Design Automation Conference*, pages 617-623, May 1987.
- [6] Yuji Kukimoto, Robert K. Brayton, and Prashant Sawkar. Delay-optimal technology mapping by dag covering. In *Proceedings of the Design Automation Conference*, 1998.
- [7] Clifford Liem, Trevor May, and Pierre Paulin. Instruction-set matching and selection for dsp and asip code generation. In *Proceedings of EDAC-ETC-EUROASIC*, pages 31-37, 1994.
- [8] Giovanni De Micheli. private communication, 1998.
- [9] Frederick Onion, Alexandru Nicolau, and Nikil Dutt. Compiler Feedback in ASIP Design. Technical report, University of California, Irvine, September 1994.
- [10] Stamatis Vassiliadis, James Phillips, and Bart Blaser. Interlock Collapsing ALU's. *IEEE Transactions on Computers*, 42:825-839, July 1993.



**CHAMELEON**  
S Y S T E M S , I N C.

## **Wireless Base Station Design Using Reconfigurable Communications Processors**

© 2000 Chameleon Systems, Inc. All rights reserved.  
Chameleon Systems is a registered trademark and  
eConfigurable and eBIOS are trademarks of  
Chameleon Systems, Inc.  
V1.0—0005

Chameleon Systems, Inc  
161 Nortech Parkway  
San Jose, CA 95134

<http://www.chameleonsystems.com>

## Introduction

---

As the communications market continues its explosive growth and rapid rate of change, equipment vendors struggle with the conflicting goals of performance, flexibility, low cost and fast time-to-market. Traditional processing approaches such as DSPs, ASICs, ASSPs and FPGAs all force the designer to sacrifice at least one of these key parameters.

A new class of processor from Chameleon Systems, the Reconfigurable Communications Processor (RCP), enables designers to meet all these goals simultaneously for multi-channel, data-processing intensive applications.

## Design Challenges

---

### *Insatiable Demand for Performance*

The explosive demand for increased communications bandwidth is forcing communication equipment vendors to focus on performance. There are several driving factors behind this demand for performance. First, there is the dramatic increase in the number of Internet users. The second driving factor is the shift in Internet usage to more bandwidth intensive applications, such voice over IP (VoIP) and streaming video.

Traditionally, these impacts were limited to fiber optic and copper networks. Today, performance demands are also impacting the wireless market as users start to implement wireless LANs and access the Internet over cellular connections. Analysts forecast the number of worldwide wireless Internet subscribers to grow from 2 million in 1999 to over 93 million in 2004, and these new users will demand the same features and bandwidth they get on their desktop computers.

The current Wireless Application Protocol (WAP) supports limited email and text-based web browsing. Emerging wireless protocols that support the features demanded by next-generation users require more processing power. A single 30kHz TDMA channel, for example, requires about 40Mips for channel filtering, equalization and modulation/demodulation. In comparison, a 1.2288 Mcps CDMA correlator serving perhaps 20 users requires about 10 GOp/sec for just the rake receiver processing.

### *Need for Flexibility*

In addition to performance considerations, equipment vendors are forced to build in flexibility to adapt to rapidly changing market requirements. Convergence of voice, data and video, changing standards, and a high demand for evolving features require the equipment vendor to build systems that are flexible and field upgradeable.

Today, there is a strong demand for multi-protocol systems that can adapt to changing traffic patterns or support multiple markets. U.S. wireless infrastructure for example, must handle analog traffic, cdmaOne and TDMA digital traffic as well as the emerging cdma2000 standard. Since no one can accurately predict the volume of each type of traffic over the next few years, vendors strive to create flexible systems that can instantaneously adapt to changing patterns. Flexibility also allows vendors to differentiate their products and create higher value using proprietary algorithms.

### ***Time-to-market Requirements***

In an environment where requirements are rapidly changing, the time required to bring new products to market can spell the difference between success and failure. Equipment vendors need an approach that allows them to quickly design, debug and verify their systems.

### ***Cost Pressures***

Despite the increased demand for new features, the per-user price of communications equipment and services will decline over the next few years. Price reductions are, in fact, necessary to bring emerging technologies into the mainstream market.

## **Choosing a Processor Technology**

In the past, equipment suppliers based their designs on Application Specific Standard Products (ASSPs) with programmable logic acting as glue. Alternatively, they employed programmable DSPs or FPGAs during early design and field trials, then ported the design to an ASIC implementation to reduce cost for high volume production.

Chameleon Systems' RCP provides suppliers with a new category of processor with the most favorable characteristics for multi-channel, data-processing intensive communications applications.

	RCP	ASIC	FPGA	DSP	ASSP
Flexibility	High	Low	High	High	None
Cost	Low	Low	High	Medium	Low
Performance	High	High	Medium	Low	High
Time-to-market	Medium	Long	Medium	Medium	Short

**Table 1 - Comparison of Available Technologies**

### ***DSP***

The biggest limitation of DSPs for multi-channel applications is performance. A single DSP does not have the bandwidth to process multiple wide data streams at speed. As a result, designers are forced to partition the system using multiple DSPs, which significantly increases design complexity and cost per channel.

### ***ASIC***

ASICs offer high performance, but take longer to design and lack the programmability required to provide adequate flexibility. Once deployed, systems built using ASICs suffer long delays and high costs for even minor changes.

### ***FPGA***

FPGAs are flexible, but cannot provide a complete solution for signal and protocol processing. To implement a complete system, they must be combined with a processor through a specially designed interface. The high-density FPGAs required to implement such a system also carry a significant cost premium.

From a performance standpoint, FPGAs are unpredictable. Carefully optimized designs are faster than DSPs. However, minor changes to the design can result in long optimization cycles to get back

to the required performance. In addition, FPGAs have bit-orientated structures that incur serious overhead in speed and gate resources when applied to wide datastream applications.

While FPGA solutions are flexible, they require hundreds of milliseconds to reprogram due to large configuration files. This prevents the designer from using the reconfigurability to increase performance and reduce cost by applying multiple algorithms to a data stream in a single chip.

#### **ASSP**

Application Specific Standard Products (ASSPs) are typically not available for emerging standards. Designers may have to wait a year or more after a standard is frozen before they can find an ASSP that fits their needs.

When ASSPs are available, they enable fast time-to-market and cost effective implementations, but offer little flexibility to the designer. With an ASSP-based design, vendors cannot use their intellectual property to differentiate their system, and have no flexibility to add features or adapt to changing standards.

#### **RCP**

Chameleon Systems' Reconfigurable Communications Processor, enables wireless base station designers to achieve a combination of low cost, fast time-to-market, high performance and complete flexibility.

The RCP provides a platform-based approach that incorporates three core architectural technologies: a complete 32-bit embedded processor subsystem, a high-performance 32-bit reconfigurable processing fabric, and eConfigurable™ Technology, Chameleon's patented instantaneous reconfigurability.

### **RCP Architecture**

Chameleon Systems' CS2112 RCP, built using a 0.25-micron CMOS process, provides 24,000 MOPs and 3,000 MMACS processing power – about ten times that of a high-performance DSP. This is enough to implement 50 channels of chip-rate processing for cdma2000 in a single device.

The following architectural features, shown in Figure 1, enable this tremendously high performance.

#### **Reconfigurable Processing Fabric**

The Reconfigurable Processing Fabric (RPF) is organized in slices, each of which can be independently reconfigured. The CS2112 includes four slices consisting of three tiles each. Each tile comprises seven 32-bit Datapath Units (DPUs), two 16x24-bit single-cycle multipliers, four Local Store Memories (LSMs) and a Control Logic Unit (CLU). A dynamic interconnect connects the modules within the Fabric.

#### **High Bandwidth Programmable I/O**

Unlike DSPs, the RCP includes four banks of programmable I/O pins that provide tremendous bandwidth. The CS2112 includes four banks of 40 I/O pins per bank, providing an aggregate I/O bandwidth of 2 GByte/sec., enabling high-performance data streaming for signal processing and protocol processing applications.

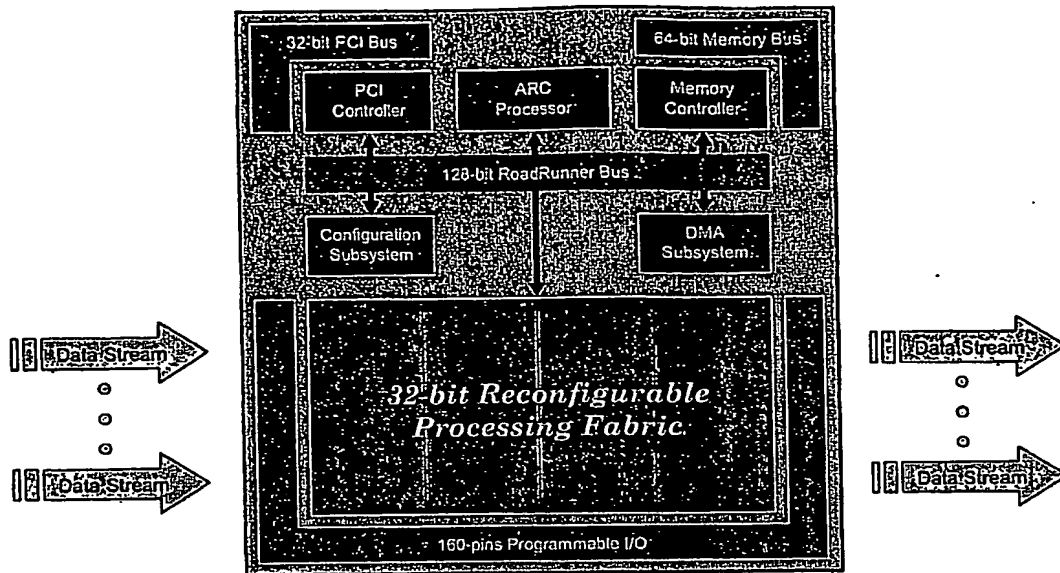


Figure 1 - Chameleon Systems' RCP

#### 32-bit ARC Processor

Optimized by Chameleon for the RCP, the 32-bit ARC Processor delivers 120 MIPS at 125 MHz. Licensed from ARC Cores Ltd., the processor employs a four-stage pipeline, 64 general-purpose 32-bit registers and a 32-bit address space. The execution unit provides fast barrel shift, fast multiply, swap, min/max and normalize operations. The processor includes a 4-KByte instruction cache and a 4-KByte data memory. A fully integrated JTAG port is also included.

#### 64-bit Memory Controller

The 64-bit Memory Controller provides a complete high-performance solution for off-chip memory. The SSRAM Controller and the SDRAM Controller both supports a 1-GByte/sec transfer rate. The Flash EEPROM Controller supports a wide variety of devices in x8 and x16 configurations with capacity from 8 to 32 Mbits.

#### DMA Subsystem

The DMA Subsystem supports 16 DMA channels, transferring data between the modules in the Embedded Processor System and to/from the Local Store Memories in the RCP. Each DMA channel can be set up as a continuously streaming buffer.

#### 32-bit PCI Controller

The 32-bit PCI (Peripheral Component Interface) Controller provides a complete interface solution to the PCI bus, supporting Master/Slave operation.

#### Flexibility Through eConfigurable Technology

Chameleon Systems' proprietary eConfigurable technology enables the entire processing fabric of the RCP to be reconfigured instantly for the ultimate in flexibility. Utilizing a background configuration plane to store the next set of configuration bits, the next configuration can be loaded

from external memory in just 3  $\mu$ secs per slice, without interfering with active processing in the fabric.

Once loaded, the configuration in the background plane can be swapped into the active plane in just one clock cycle, allowing the RCP to instantaneously adapt to changing traffic patterns or signal quality. Contents of on-chip memory are maintained during reconfiguration, allowing the user to apply multiple algorithms to the same data without using off-chip buffers.

This technology also provides the benefits of traditional reconfigurable devices, allowing systems to be upgraded in the field to enable new features or to accommodate changes in emerging protocols.

#### ***Time-to-Market***

Chameleon Systems' RCP utilizes a platform-based approach to insure the fastest possible time-to-market. Every sub-system in the device is fully integrated and pre-verified. This saves the designer the time required to implement a memory sub-system and controller, DMA engine, PCI interface, and micro-processor interfaces.

The RCP's co-design environment allows high-level algorithms to be implemented quickly using standard C and HDL languages. Complete observability and fully synchronous timing enables processor-style debug and verification of the reconfigurable processing fabric.

A break-thru eBIOS™ interface manages the interaction with the RCP, overcoming the challenges associated with custom SOC's.

eConfigurable technology also provides all the time-to-market advantages of a traditional reconfigurable device by enabling rapid in-circuit design iterations.

#### ***Cost***

For multi-channel applications, the processor cost is typically measured by dividing the overall processor cost by the number of channels supported. The resulting cost-per-channel metric allows vendors to easily compare costs of approaches supporting different numbers of channels.

Chameleon's RCP provides a very cost-effective solution for wireless base station applications. High-volume pricing is projected to reach less than \$1.00 per channel of chip-rate processing by 2H 2001 for the North American cdma2000 standard.

## **Design of a cdma2000 Base Station**

### ***Wireless Base Station Infrastructure***

The need for performance, flexibility, fast time-to-market, and low cost are especially critical in wireless base stations. Performance demands for next-generation systems are radically increased by the greater signal processing requirements of new standards, and the new features required by users. In the physical layer, for example, aggressive signal processing, such as beamforming and multi-user detection techniques are required to increase capacity and coverage.

Flexibility is required to handle the varying levels and quality of traffic from old and new protocols simultaneously. In CDMA base receivers, processing resources are allocated to received signals

In Chameleon Systems' CS2112 device, the Pseudo-Random Number Sequence Generator (PNGEN) is implemented using a pre-computed polynomial look-up-table and delay-line technique that achieves a throughput of 64 chips per clock. The signal is decoded using a match-filter technique that interpolates the received data. The derived data is then passed to a set of filter stages whose outputs are used to locate the best match based on a PN sequence.

The chip-rate and symbol-rate processors for a system with 50 user channels can be implemented in two Chameleon CS2112 RCP devices. As shown in the following figure, the chip-rate processor is implemented in one device and the symbol-rate processor is implemented in the second device.

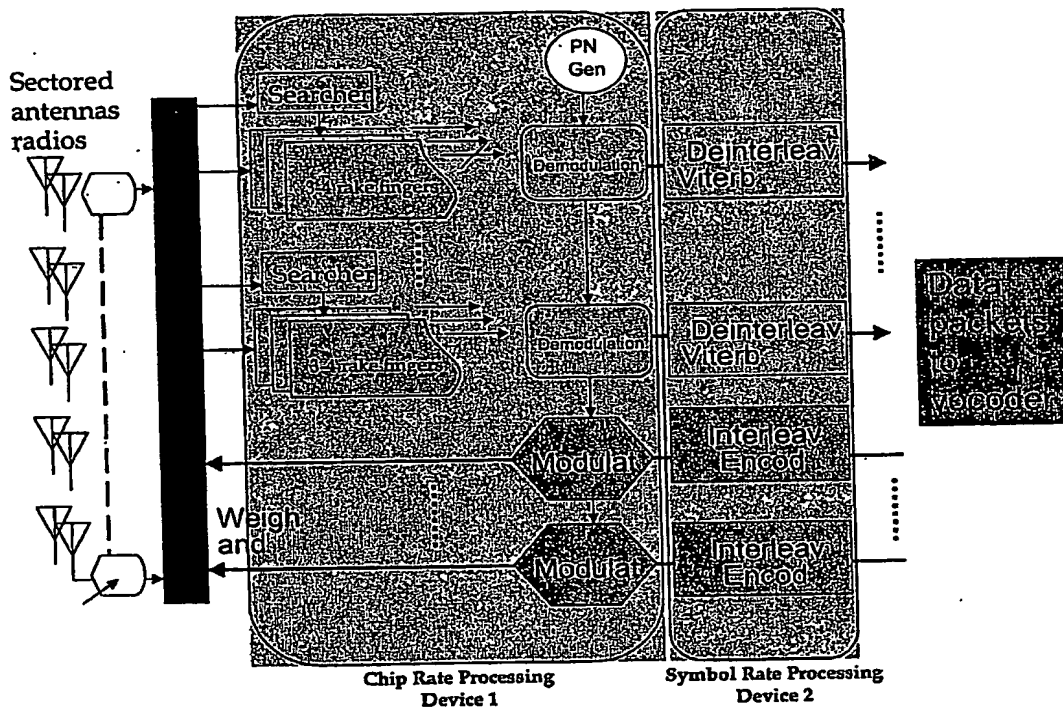


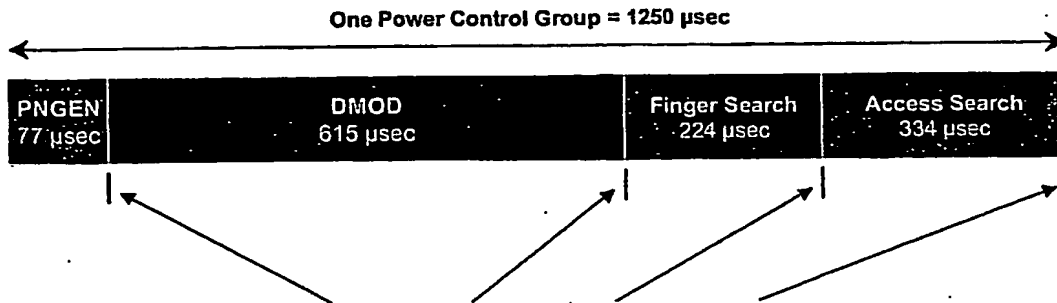
Figure 2 - cdma2000 Base Station

In Chameleon Systems' implementation of the wireless base station, eConfigurable technology is used to dramatically increase performance of the Reconfigurable Communications Processor. In this implementation, a frame of data is stored in the reconfigurable processing fabric's Local Store Memory and the device is instantaneously reconfigured to apply different algorithms to the data.

As shown in Figure 3, each frame of data, called a power control group (PCG) is 1250  $\mu$ sec long. The four algorithms that are applied to the data are loaded into the reconfigurable processing Fabric one at a time.

First, the entire Fabric is dedicated to PNGEN for 77  $\mu$ sec. While PNGEN is processing, the DMOD algorithm is loaded into the background configuration plane. In a single clock-cycle, the

entire fabric is swapped to the DMOD algorithm. While the DMOD algorithm is active, the Finger Search algorithm is loaded into the background plane.



### ***Processing Fabric Reconfigured in One Clock Cycle***

**Figure 3 – Chip-rate Processing Using eConfigurable Technology**

This continues until all four algorithms have been applied to the data. Since the entire RPF is dedicated to just one algorithm at a time, much higher performance, lower cost and lower power are achieved. In addition, there is no need to 'move' the data to the physical logic that implements the next algorithm, eliminating typical performance bottlenecks found in ASICs and FPGAs.

eConfigurable technology enables the entire chip-rate processing for a 50 channel system to be implemented in one Chameleon CS2112 device. Traditional approaches implement each of the four chip-rate processing algorithms as separate hardware modules in ASICs or FPGAs.

## **Summary**

In the race to deploy next-generation wireless protocol systems, equipment vendors must overcome the challenges of performance, flexibility, cost and time-to-market. Chameleon Systems' Reconfigurable Communications Processor meets all four of these goals enabling vendors to be first to market with the highest performance, most flexible products at the lowest cost.

We claim:

1. A method of configuring a processor to implement in hardware portions of a computer application comprising:
  - (a) providing a reconfigurable processor,
  - (b) generating a tree of nodes and arcs representing data flow and control flow of the application,
  - (c) analyzing recurrent pattern of the application including:
    - (i) clustering in clusters  $G_m^n$  recurrent sequences of instructions in code implementing the application, where  $n$  denotes the granularity level and  $m$  denotes the type of cluster at a given level of granularity,
    - (ii) rewriting the instructions by replacing at least a portion of the instructions in the code with previously identified clusters,
    - (iii) clustering in further higher level clusters  $G_m^n$  recurrent sequences in the code as rewritten,
    - (iv) rewriting the rewritten instructions by replacing at least a portion of the recurrent sequences clustered in step (c) (iii) with the further higher level clusters identified in that step,
  - (d) configuring the processor to implement at least a portion of the recurrent sequences clustered in at least one of steps (c) (i) and (c) (iii) as hardware modules,
  - (e) developing in computer memory a library of the hardware modules for use in implementing the recurrent sequences of the hardware modules.
2. The method according to claim 1, further comprising assigning weights to the nodes and arcs of the tree-generated in step (b).
3. The method according to claim 2, wherein step (d) further comprises:
  - (i) providing a partitioner to assign clustered sequences to either hardware or software, and
  - (ii) providing a router to place the hardware modules on the chip.

4. The method of claim 1, further comprising detecting conditional structures in the code implementing the instructions indicative of thread level parallelism, and providing a change implementation for causing a change from one parallel thread to another.

## Pattern Recognition Tool to Detect Reconfigurable Patterns in MPEG4 Video Processing

Ali Akoglu, Aravind Dasu, Arvind Sudarsanam, Mayur Srinivasan,  
Sethuraman Panchanathan *Fellow IEEE*  
Visual Computing and Communications Laboratory  
Arizona State University  
{ali.akoglu, dasu, arvind.sudarsanam, mayur.s, pançh}@asu.edu

### Abstract

*Current approaches towards building a reconfigurable processor are targeted towards general purpose computing or a limited range of media specific applications and are not specifically tuned for mobile multimedia applications. The increasing demand for mobile multimedia processing with stringent constraints for low power, low chip area and high flexibility at both the encoder and decoder naturally demand the design and development of a dynamically reconfigurable multimedia processor. We have performed a detailed complexity analysis of the MPEG-4 video coding mode which has illustrated the potential for reconfigurable computing. We have recently proposed a methodology for designing a reconfigurable media processor. This involves the design of a parser that identifies data/control flow graphs generated from the input assembly code of an UltraSPARC V-9 architecture; recurring pattern analyzer that uses a clustering based approach to identify specific sequences of operations that can potentially be implemented in hardware; and finally a count of such modules at every level of granularity with the associated weights based on the complexity of computation and data transfers used by partitioner and router. In this paper we then propose the design of the parser and pattern recognizer with results for detecting the reconfigurable patterns in MPEG4.*

### Keywords :

reconfigurable media processor, reconfigurable architectures, dynamic reconfiguration, hardware software co-design, hardware software partitioning, routing architecture, MPEG4, recurring pattern analyzer, mobile multimedia processing, data flow graph, control flow graph, partition

**This Page is Inserted by IFW Indexing and Scanning  
Operations and is not part of the Official Record**

## **BEST AVAILABLE IMAGES**

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☒ **BLACK BORDERS**
- ☐ **IMAGE CUT OFF AT TOP, BOTTOM OR SIDES**
- ☐ **FADED TEXT OR DRAWING**
- ☒ **BLURRED OR ILLEGIBLE TEXT OR DRAWING**
- ☒ **SKEWED/SLANTED IMAGES**
- ☐ **COLOR OR BLACK AND WHITE PHOTOGRAPHS**
- ☐ **GRAY SCALE DOCUMENTS**
- ☐ **LINES OR MARKS ON ORIGINAL DOCUMENT**
- ☒ **REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY**
- ☐ **OTHER: \_\_\_\_\_**

**IMAGES ARE BEST AVAILABLE COPY.**

**As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.**